VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace základních kódů proměnné délky

Implementation of Basic Variable-length Codes

2011 Petr Bartusek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Petr Bartusek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace základních kódů proměnné délky
Implementation of Basic Variable-length Codes

Zásady pro vypracování:

Cílem práce je nastudovat a implementovat několik základních typů kódování pro symboly/čísla, porovnat z hlediska efektivity a hlavně připravit je pro použití v kompresním frameworku.

Postupujte dle následujících bodů:
1. Nastudujte následující typy kódování:
   - Phased-In kódy,
   - Redundancy Feedback kódování,
   - Self-Delimiting kódy,
   - Huffmanovo kódování a jeho varianty.
2. Implementujte jednotlivá kódování.
3. Otestujte jejich efektivity pro různé druhy dat.
4. Sestavte moduly pro jednotlivá kódování pro použití v kompresním frameworku.

Seznam doporučené odborné literatury:

Variable-length codes for Data Compression, David Salomon, Springer, 2007
Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 19.11.2010
Datum odevzdání: 06.05.2011

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem bakalářskou práci Implementace základních kódů proměnné délky vypracoval samostatně pod vedením Ing. Jana Platoše, Ph.D. a uvedl v ní všechny použité literární a jiné odborné zdroje v souladu s právními předpisy, vnitřními předpisy Vysoké školy Báňské – Technické Univerzity Ostrava a vnitřními akty řízení Fakulty elektrotechniky a informatiky.

V Ostravě dne …....

Podpis

„Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.“

V Ostravě dne …....

Podpis

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Janu Platoši, Ph.D. za ochotu a vstřícnost, se kterou mi poskytoval cenné připomínky a odborné rady, kterými také přispěl k vypracování této bakalářské práce.

Abstrakt

Cílem mé práce bylo seznámit se s několika základními metodami kódování proměnné délky pro čísla a znaky. Poté je implementovat do kompresního frameworku a porovnat jejich efektivitu.

V zadání bylo zvoleno Phased-In kódování, Redundancy-Feedback kódování, Self-Delimiting kódování, ve kterém jsem zvolil jako zástupce Simple-9 kódování, a Huffmanovo kódování. Po dohodě s vedoucím práce bylo upuštěno od testování variant Huffmanova kódování ve prospěch pokročilého prefixového kódování proměnné délky. V jeho rámci jsem testoval Levensteinovo kódování, Elias gamma kódování, Elias omega kódování, Exponenciálně-Golombovo kódování a Fibonacciho kódování.

Při porovnávání efektivity jsem se především zaměřil na úsporu místa získanou kompresí dat. Implementace byla s výjimkou Simple-9 kódování, které může ze své definice kódovat pouze čísla do velikosti 2^{28}-1, provedena na rozsahu vstupních hodnot <1; 2^{32}>.

Porovnání efektivity kódování bylo zkoumáno na sadě testovacích souborů poskytnutých vedoucím práce.

Klíčová slova

Phased-In kódování, Redundancy-Feedback kódování, Self-Delimiting kódování, Prefixové kódování, Levensteinovo kódování, Elias gamma kódování, Elias omega kódování, Simple-9 kódování, Exponenciálně-Golombovo kódování, Fibonacciho kódování, Huffmanovo kódování.

Abstract

The aim of my work was to learn a few basic methods of variable-length coding for the numbers and the characters. Thereafter, I implemented them to the compression framework and compared their efficiency.

There were chosen Phased-In encoding, encoding Redundancy-Feedback, Self-Delimiting coding including Simple-9 coding and Huffman coding, in the assignment. In agreement with the supervisor was dropped from testing variants of Huffman coding in favor of advanced prefix variable lengths coding. Within this task, I tested Levenstein coding, Elias Gamma coding, Elias Omega coding, Exponential Golomb coding and Fibonacci coding.

During the comparison of the effectiveness, I primarily focused on saving space acquired by data compression. The implementation was performed on a range of input values <1, 2^{32}>, with the exception of the Simple-9 encoding, which can only be encoded by numbers up to 2^{28}-1, according to the definition.

The comparison of coding efficiency was examined on a set of test files provided by the supervisor.

Keywords

Phased-In coding, Redundancy-Feedback coding, Self-Delimiting coding, Prefix coding, Levenstein coding, Elias Gamma coding, Elias Omega coding, Simple-9 coding, Exponential-Golomb coding, Fibonacci coding, Huffman coding.

Seznam použitých symbolů a zkratek

HS	Huffmanův strom
HK	Huffmanovo kódování
RF	Redundancy Feedback

Seznam tabulek

Seznam výpisů zdrojového kódu

Obsah

1 Úvod

Při práci s velkým množstvím dat většinou vyvstává otázka nutnosti jejich komprese. Její správná volba nám ušetří nejen důležité místo na médiu, ale také čas potřebný k přenosu těchto dat sítí z počítače do počítače. A ačkoliv je princip některých níže popisovaných kódování znám i více než sto let, dostává se jim většího významu teprve se vzrůstajícím výkonem moderních počítačů.

Kompresi dat můžeme rozdělit na ztrátovou a bezeztrátovou. Při ztrátové kompresi dovolíme možnost ztráty části informace, proto je vhodná především pro kompresi obrázků, zvuku a videa, kde z důvodu nedokonalosti lidských smyslů nemusíme ztrátu informace postřehnout. Oproti tomu bezeztrátová komprese zachovává všechny informace obsažené v předloze a ač se sice nemůže svým kompresním poměrem rovnat kompresi ztrátové, je aplikována v případech, kdy je jakákoliv ztráta dat nepřípustná.

Tato bakalářská práce se zaměřuje výhradně na bezeztrátovou kompresi a to na kódování, u nichž je velikost binární reprezentace čísel či znaků závislá na velikosti čísla či procentuální pravděpodobnosti výskytu znaku, na rozdíl od v informatice častěji používaného kódování pevné délky.

V první kapitole představí principy všech testovaných kódování a v kapitole následující pak popíše jednotlivé kroky algoritmů na příkladech zdrojových kódů. Poslední kapitola je pak věnovány zhodnocení efektivity těchto algoritmů na různých vzorcích dat. Součástí této práce je i CD se zdrojovými kódy.

2 Principy jednotlivých kódování

V této práci se postupně setkáme se třemi přístupy kódování znaků a čísel. První přístup, který je využitelný pro čísla i znaky, je založený na statistické analýze celého textu, která je základem k nalezení optimální komprese dat. Mezi tato kódování patří Phased-In kódování, Redundancy Feedback kódování a Huffmanovo kódování.

Druhým přístupem, který je však efektivně využitelný pouze pro kompresi čísel, je tzv. prefixové kódování, které pro všechna čísla definuje odpovídající kód takový, aby kód žádného čísla nebyl prefixem kódu čísla jiného. Mezi prefixová kódování patří mimo jiné Levensteinovo kódování, Elias gamma kódování, Elias omega kódování, Exponenciálně-Golombovo kódování a Fibonacciho kódování.

Poslední v této práci popisovaný přístup je velmi odlišný od prvních dvou postupů. Self-Delimiting kódování totiž nevypočítává ani statistiku četnosti znaků ani nekóduje symboly prefixovými kódy. Toto kódování se snaží zužitkovat výhody jak kódování variabilní délky, tak i kódování fixní délky. Mezi Self-Delimiting kódování patří i níže popsané Simple-9 kódování.

Jak můžeme zjistit v tabulce 1, tak jednotlivá prefixová kódování volí pro prezentaci čísla *n* rozdílné řetězce.

n	Levensteinovo kódování	Elias Gamma kódování	Elias Omega kódování	Exponenciálně-Golombovo kódování (k=0)	Fibonacciho kódování
0	0	-	-	1	-
1	10	1	0	010	11
2	1100	010	100	011	011
3	1101	011	110	00100	0011
4	1110000	00100	101000	00101	1011
5	1110001	00101	101010	00110	00011
6	1110010	00110	101100	00111	10011
7	1110011	00111	101110	0001000	01011
8	11100000	0001000	1110000	0001001	000011
9	11100001	0001001	1110010	0001010	100011
10	11100010	0001010	1110100	0001011	010011

Tabulka 1: Jednotlivá prefixová kódování pro 0 <= n < 11

2.1 Statistické metody komprese dat

Základním principem statistických algoritmů je analýza textu, která slouží k nalezení optimální komprese dat. Algoritmy, které analyzuji níže, pracují s tzv. proměnlivou délkou kódu jednotlivých znaků. Princip tedy spočívá v nahrazení četnějších znaků kratšími binárními kódy. Lempel-Ziv algoritmy a adaptivní Huffmanovo kódování pak využívají dynamickou analýzu textu.

2.1.1 Phased-In Kódování

Phased-In kódování vzniklo jako částečný kompromis mezi kódováním fixní délky, které se nejčastěji využívá při kódování symbolů s podobnou pravděpodobností výskytu a prefixovým kódováním, které je naopak nejefektivnější při kódování symbolů s široce rozdílnou pravděpodobností výskytu. Phased-in kódování zakódovává symboly do bloků o maximálně dvou délkách.[8]

Základní charakteristika Phased-In kódování

Phased-In kódování komprimuje množinu vstupních znaků pevné binární délky do binárního kódu, jehož délka závisí na počtu různých kódovaných symbolů *n*. Délka jednoho znaku *m* je vždy rovna maximálně horní celé části dvojkového logaritmu počtu rozdílných kódovaných znaků ($\lceil \log_2 n \rceil$)[3]. Výsledek tohoto vztahu však nemusí být nutně celé číslo pro $2^{m-1} < n < 2^m$, proto musíme pro přesnou definici zavést i proměnné *P* a *p*. Proměnná *p* je definována jako $p = n\text{-}2^m$ a udává počet kódů délky *m*, které byly rozšířeny vždy na dva kódy délky *m+1* a proměnná *P*, která je rovna $P = 2^m\text{-}p$ a která udává počet kódů délky *m*. Pro lepší představu přikládám tabulky 2 a 3, které zobrazují kódování pro několik rozdílných hodnot *n*.

Výsledná efektivita Phased-In kódování je závislá na počtu různých kódovaných symbolů. Nejvyšší efektivitu poskytuje pro hodnoty *n* lehce vyšší než mocniny dvou a nejnižší efektivitu naopak pro *n* lehce nižší než mocniny dvou. Na rozdíl od Huffmanova kódování nemůže velikost zakódovaného symbolu nikdy přesáhnout původní velikost.

Další výhoda tvorby Phased-In kódů je snadné kódování a dekódování, které spočívá v postupném nahrazování symbolů svými Phased-In kódy.

n	m	$p = n\text{-}2^m$	$P = 2^m\text{-}p$
3	1	1	1
4	2	0	4
5	2	1	3
7	2	3	1
8	3	0	8
9	3	1	7

Tabulka 3: Parametry Phased-In kódů

I	n=3	4	5	7	8	9
0	0	00	00	00	000	000
1	10	01	01	010	001	001
2	11	10	10	011	010	010
3		11	110	100	011	011
4			111	101	100	100
5				110	101	101
6				111	110	110
7					111	1110
8						1111

Tabulka 2: Příklad šesti Phased-In kódů

2.1.2 Redundancy Feedback kódování

Redundancy Feedback (dále jen RF) kódování je originální algoritmus vymyšlený počátkem roku 2007 Eduardo Enrique González Rodríguezem, který jej však veřejně nepublikoval. Toto kódování využívá Phased-In kódů, ale podstatně se liší od ostatních entropických kódování[8].

Základní charakteristika Redundancy Feedback kódování

Většina entropických kódování přiřazuje symbolům s větší frekvencí četností kódy kratší a naopak symbolům s menší četností kódy delší. Algoritmus RF však v začátku všem symbolům přiřadí několik stejně velkých bloků kódů. Počet bloků přiřazený pro jednotlivé symboly je závislý na četnosti jejich výskytu. Délka všech těchto bloků je definována jako *m* a je rovna horní celé části dvojkového logaritmu počtu rozdílných symbolů ($\lceil \log_2 n \rceil$). Pro výpočet počtu přiřazených bloků jednotlivým symbolům si musíme nyní definovat několik základních proměnných. Proměnná f_i je rovna počtu výskytů symbolu *i* v textu. Proměnná *F* je rovna celkovému počtu symbolů. Nyní již můžeme počet přiřazených bloků pro jednotlivé symboly *p* definovat jako $p = f_i * 2^m / F$. Po tomto kroku má každý symbol přiřazeno *p* kódů z celkového počtu *m* a tudíž může být zakódován jakýmkoliv z těchto *p* kódů. V dalším kroku všem jednotlivým blokům dat jednoho symbolu *i* přiřadíme Phased-In kódy, které můžeme pojmenovat též jako „redundantní kódy“. Tyto vytvořené kódy jsou uloženy spolu se svými symboly a použity jak při kódování, tak i dekódování. Pro lepší porozumění tomuto procesu přikládám tabulku 4 s možným rozložením RF kódů pro $m = 2$, $f_A = 6$, $f_B = 4$, $f_C = 2$, $f_D = 4$.[3]

Kódování algoritmem RF začíná posledním kódovaným znakem a končí znakem prvním. Kódování probíhá tímto způsobem:

- Pro první symbol je vybrán první blok RF kódu

- Pro každý následující znak je vybrán ten RF kód, jehož „redundantní kód“ se shoduje se začátkem již zakódovaných dat. Tento „redundantní kód“ je z již zakódovaných dat odstraněn a na začátek těchto dat je připojen RF kód.

Dekódování začíná prvním přijatým kódem, který byl zároveň zakódovaný jako poslední, a probíhá tímto způsobem.

- Z dekódovaného proudu čteme vždy blok dat o velikosti *m*, který odpovídá symbolu *i*, poté jej odstraníme a a na začátek dekódovaného proudu přidáme „redundantní kód“.

Pro příklad uvádím tvorbu RF kódu řetězce „BCAACD“ podle symbolů uvedených v tabulce 4.

- 110 → 101|110 → 001|1|110 → 000|01|1|110 → 101|000|01|1|110 → 111|01|000|01|1|110

Kód	Symbol	Informace o redundantních bitech
000	A	0/3 → 0
001	A	1/3 → 10
010	A	2/3 → 11
011	B	0/2 → 0
100	B	1/2 → 1
101	C	0/1 → -
110	D	0/2 → 0
111	D	1/2 → 1

Tabulka 4: Osm RF kódů

2.1.3 Huffmanovo kódování

Huffmanovo kódování (dále jen HK) bylo zveřejněno jako závěr výzkumu komprese dat v článku „A Method for the Construction of Minimum redundancy Codes“ roku 1952 studentem Massechusetts Institute of Technology Davidem A. Huffmanem z Ohia. Od té doby bylo vytvořeno několik desítek algoritmů využívajících principu HK a tyto algoritmy jsou dodnes používány především jako část vícestupňové komprese dat[1]. HK patří mezi entropická [4] kódování (entropy encoding) využívající statistickou metodu komprese dat. Tím se velmi podobá aritmetickému nebo Shannon-Fanovu kódování.

Základní charakteristika Huffmanova kódování

Základním účelem HK je bezeztrátová komprese množiny vstupních znaků pevné binární délky do binárního kódu variabilní délky. Délka jednoho znaku je vždy menší nebo rovna horní celé části záporného logaritmu četnosti výskytu znaku v textu ($\lceil -\log_2 P_i \rceil$) [1]. Z tohoto vztahu vyplývá, že délka je menší pro znaky četnější a větší pro znaky méně četné. Zároveň však tato expanzivní vlastnost algoritmu způsobuje, že symboly s nízkou četností výskytu mohou být komprimovány do řetězců delších, než byly původní.

Kompresní poměr HK lze vypočítat jako podíl velikosti původních dat k velikosti komprimovaných dat. Velikost původních dat je definována jako součin počtu znaků a původní binární délky znaku. Velikost dat po komprimaci je rovna součinu počtu znaků a průměrné délky znaku. Po zkrácení počtu znaků z výše uvedeného vyplývá, že kompresní poměr je roven podílu původní binární délky znaku před komprimací a průměrné délky znaku po komprimaci. Průměrná délka znaku je definována jako součet všech součinů četností znaků s počtem jejich výskytů dělený celkovým počtem znaků.

1 Komprese JPEG, MP3, Ogg, WMA a ACC.

V roce 1959 dokázali pánové Gilbert a Moore, že binární kód vytvořený HK má nejmenší možnou průměrnou délku, přesněji, že nemůže existovat žádné kódování s menší průměrnou délkou kódu [6].

2.2 Prefixové kódování

Základním principem prefixového kódování je vytvoření prefixového kódu pro všechny symboly. Kód je prefixový právě tehdy, jestliže neexistuje žádný jiný znak, jehož kód by byl jeho předponou. Zdrojová data kódovaná prefixovým kódem jsou jednoznačně dekódovatelná dekódováním jednotlivých kódových slov během čtení zleva doprava, aniž by mezi jednotlivými symboly musely být oddělovače. Minimálním prefixovým kódem rozumíme takový kód, který obsahuje minimální možný počet znaků pro kódování. Délka kódu pro jednotlivé znaky se určuje z pravděpodobnosti jejich výskytu v kódovaném textu. Výhody prefixového kódování spočívají v možnosti kódovat data hned v průběhu čtení a také ve vysokém kompresním poměru pro data s vysokými rozdíly četností jednotlivých symbolů. Mezi prefixové kódy patří například kód UTF-8.

2.2.1 Levensteinovo kódování

Levensteinovo nebo také Levenshteinovo kódování je univerzální prefixové kódování kódující nezáporná celá čísla. Toto kódování bylo představeno Vladimirem Iosifovichem Levenshteinem v roce 1968.

Nevýhodou Levensteinova kódu je, že je vždy o jeden bit delší než kód vytvořený Elias omega kódováním. Na druhou stranu Levensteinovo kódování může zakódovat i nulu a tím dosahuje kratšího kódu, než posunuté Elias omega kódování.
Kódování probíhá od prvního znaku ke znaku poslednímu a to ve čtyřech krocích:

1. Inicializuj proměnnou *C* na 1
2. Binárně zapiš číslo bez jedné jedničky na začátku, ulož počet zapsaných bitů do *M*
3. Pokud je *M* nenulové, zvyš *C* o jedna a pokračuj v bodě 2 s *M* jako novou hodnotou
4. Zapiš na začátek *C* jedniček následovaných jednou nulou

Pro snadnější pochopení uvádím příklad zakódování čísla 17. Na začátek je zapsáno číslo 0001 jako binární reprezentace čísla 17 bez prvního bitu. Poté číslo 00 jako binární reprezentace čísla 4 bez prvního bitu vyjadřující délku z předchozího kroku. V posledním kroku číslo 0 jako reprezentaci čísla 2. Nyní 0 představující oddělovač před třemi jedničkami jako počet iterací zvýšený o 1. Výsledný řetězec se zvýrazněnými skupinami vypadá takto: 11110 0 00 0001.

Dekódování Levensteinova kódu probíhá třemi kroky:

1. Spočítej počet jedniček než narazíš na nulu a ulož do *M*, pokud se *M* rovná 0, vrať 0
2. Nastav proměnnou *N* na 1 a opakuj bod 3 (*M* - *1*) krát
3. Přečti *N* bitů a vlož na začátek 1, nastav výsledek na hodnotu těchto bitů přepsáním hodnoty předchozí

Pro snadnější pochopení uvádím příklad dekódování řetězce 111100000001 z předchozího příkladu. V prvním kroku jsou z kódu přečteny čtyři jedničky. Poté jsou z kódu třikrát čteny bity podle délky proměnné *N*. Dojde tedy postupně k načtení 0 => 2, 00 => 4, 0001 => 17. Výsledek je po posledním kroku uložen v proměnné *N*.

2.2.2 Elias gamma kódování

Elias gamma kódování je univerzální prefixové kódování kódující přirozená čísla představené Peterem Eliasem. Toto kódování je vhodné i pro případy, kdy není předem známo nejvyšší z kódovaných čísel a také pro kompresi dat, kde převažuje počet malých čísel nad čísly velkými.

Základní charakteristika Elias gamma kódování

Elias gamma kódování je primárně určeno pro kódování přirozených čísel. Jedním ze způsobů jak přidat nulu je přičítat 1 při kódování a odčítat při dekódování. Jiným způsobem je přidávat k nenulovým číslům prefix 1. Ke kódování celých čísel je možné využít bijekce, kdy se posloupnost (0, -1, 1, -2, 2, ...) postupně mapuje na (1, 2, 3, 4, 5, ...).

Elias gamma kódování by mělo být použito pouze v případě, pokud známe nebo aspoň odhadujeme, že rozložení čísel ke kodování má blízko k ideální distribuci pro daný kód. Jednoduchý výpočet zde ukazuje, že prvních n čísel je uniformně rozloženo (například, že se každé vyskytuje se stejnou pravděpodobností). Průměrná délka gama kódu se ukazuje, že je daleko větší než fixovaná délka binárního kódu stejných čísel. Délka gama kódu pro číslo i je $1+2*\lfloor \log2(i) \rfloor$. Průměrná délka gama kódů pro prvních n čísel je $a = (n+2\ \Sigma^{n}_{i=1}\ \lfloor \log2(i) \rfloor)/n$. Délka kódů fixované délky stejných čísel je $b = \lceil \log2(n) \rceil$.

Elias gamma kódování se tedy vyplatí v případech, kdy je rozložení pravděpodobností mocninného charakteru a tedy velmi zešikmené, z čehož vyplývá, že nejčastěji se vyskytuje několik málo symbolů (známé Paretovo pravidlo). Naopak v exponenciálním rozložení, kdy mají malé hodnoty podobné výskyty, je lepší použít jiné kódování např. Riceovo.

Kódování probíhá od prvního znaku ke znaku poslednímu a to ve dvou krocích:

1. Vypočítej binární reprezentaci čísla
2. Na začátek připiš tolik nul, kolik bitů jsi napsal v bodě 1 mínus 1

Pro snadnější pochopení uvádím příklad zakódování čísla 17. V prvním kroku číslo 17 převedeme na do jeho binární podoby, která se rovná 10001 a v kroku druhém na začátek připíšeme (5-1) nul. Výsledkem je řetězec 000010001.

Dekódování Elias gamma kódu probíhá dvěma kroky:

1. Přečti počet nul před první jedničkou
2. Přečti počet bitů podle čísla z prvního kroku plus jedna, tento řetězec je binárním zápisem dekódovaného čísla

Pro snadnější pochopení opět uvádím příklad dekódování řetězce 000010001 z předchozího příkladu. V prvním kroku dojde k přečtení 4 nul, takže v kroku druhém dojde k načtení 5 dalších bitů. Výsledkem je tedy číslo 17, protože posledním krokem došlo k načtené jeho binární reprezentace 10001.

2.2.3 Elias omega kódování

Elias omega kódování bylo, stejně jako Elias gamma, vyvinuto Peterem Eliasem pro kódování přirozených čísel. Toto univerzální kódování narozdíl od Elias gamma kódování spočívá v rekurzívním kódování prefixu. Je tedy občas známo i pod názvem rekurzívní Elias kódování.

Základní charakteristika Elias omega kódování

Podstatou Elias omega kódování je vložit na začátek řetězce délku n řetězce jako první skupinu, poté délku délky jako další skupinu a tak pokračovat dokud není poslední délka velikosti 2 nebo 3, čímž se nám již vejde do dvou bitů. K rozlišení jednotlivých kódů slouží jako oddělovač znak nuly.

Při pohledu na tabulku 1 s Elias gamma a omega kódy stejně jako z pravidla jejich tvorby můžeme vyvodit, že jejich délky fluktuují. Délka kódu při zvyšujícím se n narůstá pomalu, ale při překročení hranice 2^{2^k}, kde k je přirozené číslo, se přidá nová skupina a délka naroste skokově o několik bitů. Pro k = 1, 2, 3, 4 jsou tedy hranice 4, 16, 256 a 65536. Protože skupiny délek jsou ve formě "délka,", "log(délka)", "log(log(délka))", atd., Elias omega kódování je někdy nazýváno také jako rampově logaritmické kódování.[8]

Kódování probíhá od prvního znaku ke znaku poslednímu a to rekurzivně ve třech krocích:

1. Vlož znak "0" na konec řetězce

2. Pokud je kódované číslo 1 tak konec, jinak na začátek vlož binární reprezentaci aktuálního čísla
3. Opakuj předchozí krok s číslem reprezentujícím počet zapsaných číslic z bodu 2 sníženého o 1

Pro lepší vysvětlení opět uvádím příklad s kódováním čísla 17. V prvním kroku vložíme na konec 0 a před ní 10001. Tato binární skupina je dlouhá 5 bitů, takže se před ní vloží skupina nová 100, reprezentující číslo 4 (5-1). Tato skupina je dlouhá 3 bity, takže krok 2 opakujeme a vložíme ještě skupinu 10 reprezentující číslo 2 (3-1). Nyní se rekurze zastaví a vrátí se nám výsledný řetězec, který se zvýrazněnými skupinami vypadá takto 10 100 10001 0.

Dekódování probíhá rekurzivně také ve třech krocích:

1. Nastav proměnnou *N* na 1
2. Přečti *N+1* znaků, pokud následuje 0, tak je proměnná *N*
3. Převeď přečtené znaky do čísla a ulož do *N*, pokračuj bodem 2

Jako příklad dekódování nám poslouží řetězec 10100100010 z předchozího příkladu. V prvním kroku je proměnná *N* nastavena na 1 a v kroku druhém dojde k přečtení dvou bitů, ve kterých se nachází "10", což vyjadřuje číslo 2. Dále jsou přečteny 3 bity, ve kterých je uložena hodnota 4. V dalším kroku je přečteno 5 bitů, které vyjadřují hodnotu 17. Kdyby nenásledoval znak nuly, přečetlo by se v tomto kroku 17 dalších bitů. V tomto případě však představuje číslo 17 výsledek.

2.2.4 Exponenciálně-Golombovo kódování

Exponenciálně-Golombovo kódování řádu *k* je typ univerzálního kódu, parametrizovaného celým nezáporným číslem *k*. K zakódování při $k = 0$ stačí zapsat číslo v binárním tvaru zvýšeného o 1 a přidat na začátek tolik nul, kolik bitů jsme zapsali mínus 1. Zajímavostí je, že Exponenciálně-Golombovo kódování je identické s Elias gama kódováním na daném číslu +1. Z této vlastnosti vyplývá, že na rozdíl od gama kódování dokáže kódovat i nulu.

Exponenciálně-Golombovo kódování pro $k = 0$ je použito ve standardu video komprese H.264/MPEG-4 AVC, kde je navíc i variace pro kódování celých čísel přiřazením hodnoty 0 k binárnímu kódovanému slovu '0' a přiřazením následujících kódových slov ke vstupním hodnotám zvyšující se důležitosti. Dále je použito například v Diracově video kodeku.

Kódování probíhá od prvního znaku ke znaku poslednímu a to ve třech krocích:

1. Vezmi binární reprezentaci čísla kromě posledních *k* znaků a přičti k ní 1
2. Na začátek připiš tolik nul, kolik bitů jsi napsal v bodě 1 mínus 1
3. Připiš posledních *k* bitů původního čísla

Pro lepší vysvětlení uvádím příklad s kódováním čísla 17 s parametrem $k = 2$. Binární reprezentace čísla 17 je 10001 a tudíž v prvním kroku dojde k rozdělení řetězce na 100 a 01 a přičtení 1 k číslu 100, které se následně zapíše. Na začátek se v druhém kroku připíšou dvě nuly (délka zápisu – 1). Ve třetím kroku se řetězec 01 připíše na konec řetězce. Nyní výsledný řetězec se zvýrazněnými skupinami vypadá takto: 00 101 01

Dekódování Exponenciálně-Golombovo kódu probíhá ve třech krocích:

1. Přečti počet nul před jedničkou
2. Z předchozího počtu přečti binární reprezentaci čísla a odečti 1
3. Z předem známé hodnoty *k* se přiřetězí dalších *k* bitů, což je výsledek

Jako příklad uvedu dekódování řetězce 0010101 z minulého příkladu. Krokem prvním načteme dvě nuly, které symbolizují číslo dvě a proto v kroku třetím přečteme z kódu 3 znaky 101 a snížíme je o jedničku, čímž nám vznikne číslo 100. K tomuto řetězci přidáme ještě ve třetím kroku z kódu počet znaků roven parametru *k*. Tím nám vznikne řetězec 10001, který je binární hodnotou čísla 17.

2.2.5 Fibonacciho kódování

Fibonacciho kódování je postaveno na Fibonacciho posloupnosti, kterou na přelomu 12. a 13. století publikoval italský učenec Leonardo Pisano Bigollo známější spíš jako Leonardo Fibonacci. Později Eduard Zeckendorf objevil, že pomocí součtu konečného počtu spolu nesousedících čísel z fibonacciho posloupnosti, lze vyjádřit jakékoliv přirozené číslo.

Fibonacciho kódování kóduje čísla do binárních bloků zakončených znaky "11", protože máme jistotu, že nikde jinde se v kódu nevyskytnou dvě jedničky vedle sebe. Číslo *N* je pak složeno z posloupnosti Fibonacciho čísel označených jako 1 a zbytku označených jako 0. Mějme následující tabulku 5 Fibonacciho posloupnosti:

Fibonacciho posloupnost	0	1	1	2	3	5	8	13	21	34	55
Číslo Fibonacci posloupnosti	F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)	F(10)

Tabulka 5: Prvních deset čísel Fibonacciho posloupnosti

Číslo 65 pak zakódujeme jako 55+8+2, takže jako Fibonacciho čísla F(10), F(6), F(3). Algoritmus Fibonacciho kódování značí pozice od čísla F(2). Výsledkem je posloupnost 010010001, ke které je následně přidána navíc ukončující 1.

Kódování probíhá od prvního znaku ke znaku poslednímu a to ve čtyřech krocích:

1. Do proměnné N přiřaď kódované číslo
2. Najdi nejvyšší Fibonacciho číslo, které je menší nebo rovno proměnné *N*, a odečti jej od *N*. Zbytek po této operaci přiřaď do proměnné *N*
3. Pokračuje se s odečítáním až do nuly. Za každé odečtené *i*-té Fibonacciho číslo je zapsána 1, za nevyužité je zapsána 0
4. Připiš na konec připiš 1

Dekódování Fibonacciho kódu probíhá takto:

1. Přeskoč bity až k sekvenci "11". Přeskočené bity vynásob postupně hodnotami Fibonacciho posloupnosti a sečti

Charakteristika Fibonacciho kódování

Dá se ukázat, že takto kódované číslo je unikátní a nemůže dojít ke kolizi. Fibonacciho kódování je ukázkou sebesynchronizujícího kódování, je tedy jednodušší jej obnovit z poškozeného proudu dat. U většiny univerzálních kódů způsobí změna jediného bitu zničení celé sekvence. S Fibonacciho kódováním změněný bit může způsobit přečtení jednoho bloku jako dvou nebo dvou bloků jako jednoho, ale díky znakům "11" se další propagace chyby zastaví.

2.3 Self-Delimiting kódování

Self-Delimiting kódování je přístup velmi odlišný jak od prefixového kódování, tak i od statického či entropického kódování.[7] Toto kódování totiž nepracuje ani s četností znaků, ani s prefixovými kódy, ale definuje si blok velikosti *n* bitů, který se skládá ze dvou částí. Za prvé z bloku dat o velikosti *m*, ve kterém jsou uložena zakódovaná čísla a za druhé z hlavičky o velikost *n-m*, která definuje počet a délku jednotlivých zakódovaných čísel, popřípadě i nejnižší kódované číslo. Toto kódování dosahuje vysoké efektivity především při hustém rozložení hodnot kódovaných čísel nebo pro kódování malých čísel. Mezi Self-Delimiting kódování patří tzv. Simple kódování jako Simple-9, Simple-10 a Simple-16.[8]

2.3.1 Simple-9 kódování

Simple-9 kódování předpokládá datový stream složený z velkého počtu malých přirozených čísel, které kóduje po blocích s pevnou délkou. Toto kódování definuje celkový blok o velikosti $n = 32$ bit, datový blok o velikosti $m = 28$ bitů, který může být rozdělen na k-bitové fragmenty a hlavičku o velikosti 4 bitů, která určuje velikost parametru k. Jelikož číslo 28 má vysoký počet dělitelů, dostáváme celkem 9 možností rozdělení, které vystihuje tabulka 6, a tyto možnosti lze zakódovat do 4 zbývajících bitů hlavičky. Zatímco například číslo 27 má jen 4 dělitele a kód by nebyl tak variabilní, číslo 30 má sice dělitelů 8, ale tyto varianty nelze zakódovat do zbylých dvou bitů hlavičky.

Hlavička	Počet kódovaných čísel	Délka kódu	Nepoužité bity
0000	28	1	0
0001	14	2	0
0010	9	3	1
0011	7	4	0
0100	5	5	3
0101	4	7	0
0110	3	9	1
0111	2	14	0
1000	1	28	0

Tabulka 6: Rozložení 32 bitového bloku kódováním Simple-9

Kódování probíhá od prvního znaku ke znaku poslednímu a to ve čtyrech krocích:

1. Nastav si maximální bitovou velikost čísla *M* na 1
2. Porovnej bitovou velikost čísla s proměnnou *M* a větší hodnotu ulož zpět do *M*. Číslo si ulož na do fronty.
3. Pokud je hodnota *M* krát délka fronty menší než 28, pak opakuj krok 2.
4. Ulož hlavičku podle proměnné *M*. Všechna čísla z fronty kromě posledního ulož do binárního kódu.

Pro snadnější pochopení přikládám příklad zakódování řady čtrnácti přirozených čísel 4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12 a 20. Tato čísla lze sdružit do dvou skupin po devíti a po pěti číslech. V první skupině nám velikost datových segmentů určuje nejvyšší číslo 7, které lze ještě pořád zapsat 3 bity. Dohromady tedy těchto devět čísel zapíšeme do 27 bitů a jeden bit nám zůstane nevyužit. Číslo 13 však už zabere bity 4 a pokud po něm budou následovat i další čísla, pak bude zakódováno do 4 bitů. Algoritmus Simple-9 tedy postupně přibírá čísla ze čteného streamu, dokud je lze zakódovat do 28 bitů. Výše uvedený řetězec je tedy zakódován do těchto dvou skupin: 0010|011|101|000|000|010|100|000|110|000|0 a 0100|01100|10011|00000|01011|10011|000. První čtyři bity představují hlavičku podle tabulky, poslední bit v první skupině a poslední tři bity ve druhé skupině jsou "nepoužité" bity a slouží jen jako výplň, aby celkový blok měl délku 32 bitů.

Dekódování streamu probíhá o poznání jednodušeji pouze ve dvou krocích:

1. Načti 32 bitů a z prvních 4 bitů zjisti, po kolika bitech *M* budeš číst datový blok
2. Čti datový blok *M* bitech.

3 Implementace

Implementace a tvorba kompresního frameworku probíhala ve vývojovém prostředí Microsoft Visual Studio 2010 v programovacím jazyce C# 4.0. Všechna kódování jsou uložena ve svých vlastních projektech a poskytují statické metody *Encode(Stream stream)* a *Decode(Stream stream)* nebo *EncodeOnlyNumbers(Stream stream)* a *DecodeOnlyNumbers(Stream stream)*. Metody *Encode*, *Decode* a *EncodeOnlyNumbers*, *DecodeOnlyNumbers* jsem implementoval s ohledem na možnosti čtení vstupních dat, a proto dokáží číst nejen ze souborů, ale také se síťových streamů.

Pro lepší separaci kódování a ostatních pomocných funkcí a metod je součástí kompresního frameworku i projekt Utilities, který poskytuje osm statických pomocných metod. Součástí zdrojových kódu jsou i komentáře sloužící ke snadnějšímu porozumění principů kódování.

Ukázky zdrojových kódů uvedené v tomto textu se nemusí plně shodovat se skutečnými zdrojovými kódy v případě, kdy tím bylo dosaženo lepší čitelnosti kódu a dodržení principu stručnosti a přiměřeného rozsahu, odstraněním pomocných, na algoritmu nezávislých částí kódu. Pro plné porozumění proto doporučuji prozkoumat zdrojový kód přímo v **.cs* souborech a také se podrobněji seznámit se statickými metodami třídy *UtilityClass*.

3.1.1 Phased-In Kódování

Implementace Phased-In kódování se skládá ze tří statických metod. Metody *Encod*e a *Decode* provedou zakódování a dekódování dostupného streamu. Obě metody využívají privátní metodu *countVariables*, která z parametru *n* vypočte parametry *m*, *p* a *P*.

```
private static void countVariables(int n, out int m, out int p,
out int P)
{
   m = Convert.ToInt32(Math.Floor(Math.Log(n, 2)));
   p = n - Convert.ToInt32(Math.Pow(2, m));
   P = Convert.ToInt32(Math.Pow(2, m)) - p;
}
```

Zdrojový kód 1: Metoda pro výpočet proměnných *m, p, P*

Kódování

Prvním krokem kódování je načtení vstupní množiny dat a jejich uložení do pole bajtů. Dalším krokem algoritmu je uložení všech různých symbolů do kolekce a zjištění parametru *n*, který je roven velikosti kolekce. Poté jsou metodou *countVariables* vypočteny všechny parametry *m*, *p*, a *P*. Dalším krokem metody je vytvoření binárních Phased-In kódů a jejich uložení do kolekce k odpovídajícím symbolům. Poté již dochází k vytvoření hlavičky kódovaného streamu z parametru *n-1* (abychom mohli zakódovat všech maximálních 256 znaků) a všech kódovaných symbolů. V následujícím kroku dochází ke kódování postupně všech symbolů směrem od prvního k poslednímu, nalezením jejich odpovídajícího Phased-In kódu v kolekci. Postupně je také bitový kód převáděn na bajtový a ten je následně ukládán do pole bajtů. V posledním kroku je doplněna hlavička hodnotou obsahující začínající počet prázdných bitových nul.

```
List<byte> encodedText = new List<byte>();
int numberOfEmptyBits;

byte[] header = new byte[0];
byte[] encodedNumbers =
UtilityClass.ReadDataFromEncodedFile(stream, 0, ref header);
```

```
foreach (byte symbol in encodedNumbers)
{
   if (!symbolsTable.Contains(symbol.ToString()))
          symbolsTable.Add(symbol.ToString(), 0);
}

n = symbolsTable.Count;

countVariables(n, out m, out p, out P);
// Výpočet Phased-In kódů
for (int i = 0; i < n; i++)
{
   string symbol;
   if (i < P)
          symbol = UtilityClass.PaddedBin(i, m);
   else
          symbol = UtilityClass.PaddedBin(P + (i - P) / 2, m) +
((i - P) % 2).ToString();

   symbolsTable[i] = symbol;
}
// Zapsání N do hlavičky
encodedText.Add((byte)(symbolsTable.Count - 1));
// Zapsání všech znaků do hlavičky
foreach (DictionaryEntry i in symbolsTable)
{
   encodedText.Add(Convert.ToByte(i.Key));
}

lengthOfHeader = encodedText.Count;

foreach (byte symbol in encodedNumbers)
{
   sb.Append(symbolsTable[symbol.ToString()]);

   while (sb.Length >= 8)
          encodedText.Add(UtilityClass.MakeBinaryStringFrom-
Beginning(sb));

}
encodedText.Insert(lengthOfHeader, (byte)numberOfEmptyBits);

return encodedText.ToArray();
```

Zdrojový kód 2: Statická metoda *Encode* Phased-In kódování

Dekódování

Dekódování začíná načtením hlavičky kódovaného streamu, ze které přečteme parametr *n*, ke kterému se přičte jednička, a následně všechny kódované symboly. Tyto symboly jsou následně uloženy do kolekce. Z parametru *n* jsou pomocí metody *countVariables* vypočteny parametry *m*, *p* a *P*. Dalším krokem dekódování je vytvoření Phased-In kódů pro všechny symboly v kolekci. Poté je pomocí statických metod třídy *UtilityClass* zakódovaný stream převeden na pole bajtů. Následně je toto pole převedeno na řetězec obsahující bitovou reprezentaci zakódovaného streamu. Z tohoto řetězce jsou odstraněny nuly, které nekódují žádný symbol, a tím je připraven pro následné dekódování. Dekódování probíhá v cyklech, kdy je za prvé načteno *m* znaků řetězce a pokud je hodnota těchto znaků menší nebo rovna *P*, pak je hledaný symbol nalezen v kolekci podle těchto *m* znaků. Pokud je však hodnota

těchto *m* znaků větší než *P*, pak dojde k načtení ještě jednoho znaku a hledaný symbol je nalezen v kolekci podle těchto *m+1* znaků. Řetězec obsahující bitovou reprezentaci zakódovaného streamu je na konci cyklu zkrácen o *m* nebo *m+1* znaků.

```
List<byte> decodedText = new List<byte>();
BinaryReader br = new BinaryReader(stream);
int numberOfSymbols = br.ReadByte() + 1;

StringBuilder sb = new StringBuilder();
byte[] header = new byte[numberOfSymbols + 1];

byte[] encodedNumbers = UtilityClass.ReadDataFromEncoded-
File(stream, numberOfSymbols + 1, ref header, 1);

string decodedString =
UtilityClass.ConvertDataToStringOfBits(encodedNumbers);

// Smaže první nepoužité nuly
int numberOfZeros = header[numberOfSymbols];

sb.Append(decodedString = decodedString.Substring(0,
decodedString.Length - numberOfZeros));

n = numberOfSymbols;
// Výpočet parametrů m, p a P
countVariables(n, out m, out p, out P);
// Tvorba Phased-In kódů
for (int i = 0; i < n; i++)
{
   string symbol;
   if (i < P)
          symbol = UtilityClass.PaddedBin(i, m);
   else
          symbol = UtilityClass.PaddedBin(P + (i - P) / 2, m) +
((i - P) % 2).ToString();

   symbolsTable.Add(symbol, header[i]);
}

string symbolString;

while (sb.Length >= m)
{
   // Uřízne posledních m znaků řetězce
   symbolString = sb.ToString().Substring(0, m);
   if (Convert.ToInt32(symbolString, 2) >= P)
          symbolString = sb.ToString().Substring(0, m + 1);
   //Vyhledá symbol a převede na byte hodnotu
   byte symbol = Convert.ToByte(symbolsTable[symbolString]);
   decodedText.Add((byte)symbol);
   sb.Remove(0, symbolString.Length);
}
return decodedText.ToArray();
```

Zdrojový kód 3: Statická metoda *Decode* Phased-In kódování

3.1.2 Redundancy Feedback kódování

Implementace Redundancy Feedback kódování se skládá ze šesti statických metod. První z nich je metoda *Encode*, která provádí kódování symbolů pomocí metod *FillTable, FindSymbol, Match* a *PhaseIn*, a metody Decode, která provádí dekódování dostupného streamu.

Metoda *FillTable* slouží k vytvoření pole *RFTable* tříd, podle kterých je poté stream kódován nebo dekódován. Počet těchto tříd je roven 2^m. Každý znak *i* z proměnné *chars[i]* je obsažen ve *RFTable* třídách *probs[i]* krát. Každá třída obsahuje samotný kódovaný znak, RF kód a tzv. „Redundantní kód“, který odpovídá Phased-in kódu a je vypočten pomocí metody *PhaseIn*.

```
private static RFTable[] FillTable(short[] probs, char[] chars,
int m)
{
   int[] tmp_probs = new int[probs.Length];
   int[] sprobs = new int[probs.Length];
   RFTable[] rftable = new RFTable[(int)Math.Pow(2, m)];
   Array.Copy(probs, sprobs, probs.Length);
   Array.Copy(probs, tmp_probs, probs.Length);

   int k = 0;
   for (int i = 0; i < Math.Pow(2, m); i++)
   {
          int tmp_m = (int)Math.Floor(Math.Log(sprobs[k], 2));
          int p = sprobs[k] - (int)Math.Pow(2, tmp_m);
          int P = (int)Math.Pow(2, tmp_m) - p;

          rftable[i] = new RFTable();
          rftable[i].code = UtilityClass.PaddedBin(i, m);
          rftable[i].symbol = chars[k];
          rftable[i].redundancy = PhaseIn(sprobs[k] -
tmp_probs[k], tmp_m, P);

          tmp_probs[k]--;
          if (tmp_probs[k] == 0)
                k++;
   }
return rftable;
}

// spočítá phase-in hodnotu
private static string PhaseIn(int i, int m, int P)
{
   string code = "";

   if (m == 0)
          return "";

   if (i >= P)
          code = UtilityClass.PaddedBin(P + (int)((i - P) / 2),
m) + Convert.ToString((i - P) % 2);
   else
          code = UtilityClass.PaddedBin(i, m);

   return code;
}
```

Zdrojový kód 4: Pomocné metody *FillTable* a *PhaseIn*

Kódování

Kódování algoritmem Redundancy Feedback probíhá v několika krocích. V prvním kroku dojde k sestavení tabulky znaků a jejich četností z načítaného streamu. U vytváření pole četností si musíme dát pozor, aby pravděpodobnost nějakého čísla nebyla nula a aby se součet všech pravděpodobností rovnal 2^m. Poté je stanoveno číslo m. Po několika testováních jsem dospěl k názoru, že nejefektivněji algoritmus kódoval čísla pro $m = \lceil log_2 n \rceil$, kde n je počet rozdílných kódovaných symbolů. V dalším kroku metoda *FillTable* sestaví pole tříd, ze kterého bude v dalších krocích sestavovat výsledný kód. Nakonec ještě před samotným kódováním zapíšeme do streamu hlavičku s číslem m a všemi kódovanými znaky a jejich pravděpodobnostmi.

```
byte[] encodedNumbers =
UtilityClass.ReadDataFromEncodedFile(stream, 0, ref header);
foreach (byte symbol in encodedNumbers)
{
   if (symbolsTable.Contains(sSymbol))
          symbolsTable[sSymbol] = Convert.ToInt32(symbols-
Table[sSymbol]) + 1;
   else
          symbolsTable.Add(sSymbol, 1);
 }

m = 1 + (int)Math.Ceiling(Math.Log(symbolsTable.Count, 2));

probs = new short[symbolsTable.Count];
chars = new char[symbolsTable.Count];

// Spočítá frekvence znaků
int i2 = 0;
IDictionaryEnumerator enumerator = symbolsTable.GetEnumerator();

int count = encodedNumbers.Length;
while (enumerator.MoveNext())
{
   probs[i2] = (short)Math.Round(Convert.ToInt32(enumerator.-
Value) * Math.Pow(2, m) / count);
   chars[i2] = (char)Convert.ToByte(enumerator.Key);
   i2++;
}

rftable = FillTable(probs, chars, m);
List<byte> encodedText = new List<byte>();
StringBuilder sb = new StringBuilder();
int lengthOfHeader;
int countOfZeros = 0;

// Zapsání N do hlavičky
encodedText.Add((byte)(chars.Length - 1));
// Zapsání všech znaků do hlavičky
for (int i = 0; i < chars.Length; i++)
{
   encodedText.Add((byte)chars[i]);
   encodedText.Add((byte)probs[i]);
}
```

Zdrojový kód 5: Zjištění pole četností RF kódování a zápis hlavičky

Samotné kódování algoritmem RF probíhá voláním metody *FindSymbol* pro pole bajtů a to směrem od posledního bajtu k prvnímu bajtu. V metodě *FindSymbol* dochází k hledání znaku, jehož „redundantní kód“ se shoduje se začátkem již zakódovaných dat. Poté je tento „redundantní kód“ z již zakódovaných dat odstraněn a na začátek těchto dat je připojen RF kód.

```
for (int i = encodedNumbers.Length - 1; i >= 0; i--)
{
   FindSymbol(encodedNumbers[i], sb.ToString(), rftable, sb);

   while (sb.Length >= 8 + m)
   {
          encodedText.Insert(lengthOfHeader,
UtilityClass.makeBinaryStringFromEnd(sb));
   }
}

// Zapíše do hlavičky počet nul
encodedText.Insert(lengthOfHeader, Convert.ToByte(countOfZeros));

return encodedText.ToArray();

// najde vhodný blok pro symbol c podle redundancí
private static void FindSymbol(byte c, string encoded, RFTable[]
rftable, StringBuilder sb)
{
   int max = -1;
   int key = -1;
   for (int i = 0; i < rftable.Length; i++)
   {
          if (rftable[i].symbol == c)
          {
                 int r = Match(rftable[i].redundancy, encoded);
                 if (r > max)
                 {
                        max = r;
                        key = i;
                 }
          }
   }
   sb.Remove(0, max);
   sb.Insert(0, rftable[key].code);
}
```

Zdrojový kód 6: Samotné kódování Redundancy Feedbeck algoritmem a metoda *FindSymbol*

Dekódování

Dekódování RF streamu probíhá v metodě *Decode* v několik oddělených krocích. Za prvé ze streamu načteme hodnotu *m* a všechny zakódované znaky a jejich četnosti. Poté v druhém kroku voláním metody *FillTable* vytvoříme pole tříd obsahující všechny zakódované znaky, jejich RF kódy i tzv. „Redundantní kódy“. V následujícím kroku začínáme číst stream od začátku ke konci, takže od posledního zakódovaného symbolu k prvnímu tak, že dojde k načtení bloku dat o velikosti *m*, který odpovídá symbolu *i*. Poté jej odstraníme a a na začátek dekódovaného proudu přidáme „redundantní kód“.

```
string decodedString = UtilityClass.ConvertDataToString-
OfBits(encodedNumbers);

m = 1 + (int)Math.Ceiling(Math.Log(numberOfSymbols, 2));
```

```
chars = new char[numberOfSymbols];
probs = new short[numberOfSymbols];

// Načtení symbolů a jejich četností
for (int i = 0, j = 0; i < numberOfSymbols; i++, j += 2)
{
   chars[i] = (char)header[j];
   probs[i] = header[j + 1];
}

// Načte počet začátečních nul
int numberOfZeros = header[2 * numberOfSymbols];
decodedString = decodedString.Substring(numberOfZeros);

// Vytvoří tabulku četností
rftable = FillTable(probs, chars, m);

//NewReadDecodedFile(fs);
string symbolString;

while (sb.Length >= m)
{
   symbolString = sb.ToString().Substring(0, m);
   //Vyhledá symbol a převede na byte hodnotu
   for (int j = 0; j < rftable.Length; j++)
   {
          if (symbolString.Equals(rftable[j].code))
          {
                 symbolsString.Add((byte)rftable[j].symbol);
                 sb.Remove(0, m);
                 sb.Insert(0, rftable[j].redundancy);
                 break;
          }
   }
}
return symbolsString.ToArray();
```

Zdrojový kód 7: Metoda *Decode* Redundancy Feedback kódování

3.1.3 Huffmanovo kódování

Implementace HK se skládá ze dvou samostatných částí, a to z komprese a dekomprese dat. Tyto části však nejsou striktně definovány. V důsledku toho existuje několik algoritmů (Adaptivní Huffmanovo kódování [3], Kanonické Huffmanovo kódování [1], atd. [2]), které řeší implementaci rozdílně. Tím vzniká také rozdílný výsledný binární kód komprimovaných dat. V následujících podkapitolách analyzuji podrobněji implementaci obou samostatných částí HK.

Komprese

Prvním krokem komprese HK je načtení vstupní množiny dat ze souboru a následné vytvoření tabulky četností všech symbolů ve této množině. Četnost symbolů je ukládána do pole o indexu, který odpovídá umístění symbolu v ANSI tabulce.

```
byte[] buffer = new byte[buffer_size];
int[] frequency = new int[256];
int buffer_read;
// Vypočtení četností
do
```

```
{
   buffer_read = fi.Read(buffer, 0, buffer_size);
   for (int i = 0; i < buffer_read; i++)
          frequency[(byte)buffer[i]]++;
}
while (buffer_read > 0);
```

Zdrojový kód 8: Načtení a vytvoření tabulky četností

Poté je z tabulky četností vytvořen Huffmanův strom (dále jen HS) voláním metody *CreateTree* tak, že v prvém kroku je pro každý symbol obsažený v tabulce četností vytvořen uzel. Tento uzel obsahuje symbol, četnost tohoto symbolu a další pomocné proměnné. V druhém kroku jsou nalézány vždy dva stromy s nejnižší četností v kořenu, z nichž je vytvořen strom nový. Tento krok se opakuje do doby, kdy již existuje pouze jeden strom. Nový strom vzniká vytvořením nového uzlu a následným navázáním dříve nalezených stromů na něj. Četnost nového stromu je rovna součtu četností všech listů stromu. Z tohoto postupu je patrné, že HK skládá strom od listů ke kořenu a tím se liší od Shannon-Fanova kódování, které naopak skládá strom od kořenu k listům.

```
// Vložení znaku do uzlu
for (int i = 0; i < frequency.Length; i++)
{
   if (frequency[i] > 0)
   {
          Node u = new Node();
          u.frequency = frequency[i];
          u.symbol = (byte)i;
          u.isLastNode = true;
          symbols[i] = u;
          huffmanTree[treeLength] = u;
          treeLength++;
   }
}

// Vytvoření celého stromu
int pocetZnaku = treeLength;
for (int i = 0; i < pocetZnaku - 1; i++)
{
   int min1_index = -1;
   int min1_value = int.MaxValue;
   // Nalezení první nejnižší četnosti
   for (int j = 0; j < treeLength; j++)
   {
          if (huffmanTree[j].nodeParent == null &&
huffmanTree[j].frequency <= min1_value)
          {
                 min1_index = j;
                 min1_value = huffmanTree[j].frequency;
          }
   }
   // Nalezení druhé nejnižší četnosti
   int min2_index = -1;
   int min2_value = int.MaxValue;
   for (int j = 0; j < treeLength; j++)
   {
          if (j != min1_index && huffmanTree[j].nodeParent ==
null && huffmanTree[j].frequency <= min2_value)
          {
                 min2_index = j;
```

```
                min2_value = huffmanTree[j].frequency;
            }
        }

        // Spojí hodnoty novým uzlem
        Node u = new Node();
        u.frequency = min1_value + min2_value;
        u.nodeChilds[0] = huffmanTree[min1_index];
        u.nodeChilds[0].nodeParent = u;
        u.nodeChilds[1] = huffmanTree[min2_index];
        u.nodeChilds[1].nodeParent = u;
        u.isLastNode = false;
        huffmanTree[treeLength++] = u;
    }

    //Uložení špičky stromu
    treePeak = huffmanTree[0];
    while (treePeak.nodeParent != null)
        treePeak = treePeak.nodeParent;
```

Zdrojový kód 9: Vytvoření Huffmanova stromu pomocí metody *CreateTree*

Z výše uvedeného vyplývá, že není řešena situace, kdy více než dva kořeny stromů mají stejnou četnost výskytu. Vznik HS tedy není unikátní a dokonce stejné znaky se stejnými četnostmi mohou z důvodu libovolné volby stromů vytvořit několik rozdílných HS. Průměrná délka znaků však bude stejná. Z těchto stromů je optimální ten, který má nejmenší odchylku od průměrné velikosti. Toho lze dosáhnout skládáním stromů s nejmenší a největší výškou[1].

```
// Zakódování řetězce
for (int a = 0; a < length && a < input.Length; a++)
{
    byte symbol = input[a];

    //nalezeni uzlu s pozadovanym znakem
    Node position = symbols[symbol];

    //nalezeni cesty ke korenu (tedy vsech bitu)
    BitArray bitBuffer2 = new BitArray(256);
    int bitBuffer2_length = 0;
    if (position.nodeParent == null)
    {
            //strom obsahuje jedinný uzel
            bitBuffer2[bitBuffer2_length++] = false;
    }
    else
            do
            {
                    bitBuffer2[bitBuffer2_length++] =
(position.nodeParent.nodeChilds[1] == position);
                    position = position.nodeParent;
                    //pozn: bity jsou opacne, nez je bezne (strom je
prochazen od konce)
            }
            while (position.nodeParent != null);

            // Přesunout do hlavního bufferu
            for (int i = bitBuffer2_length - 1; i >= 0; i--)
            {
                    bitBuffer[bitBuffer_length++] = bitBuffer2[i];
```

```
            }
        }
    }
```

Zdrojový kód 10: Zakódovaní pole symbolů pomocí metody *EncodeArray*

Dalším krokem implementace HK je uložení hlavičky, která obsahuje velikost načítaného souboru a tabulku četností. Následným krokem je již procházení streamu dat metodou *EncodeArray* a pro každý symbol nalezení uzlu, ve kterém je tento symbol uložen. Nyní postupujeme Huffmanovým stromem od tohoto uzlu ke kořenu a tuto cestu ukládáme tak, že pokud byl uzel vlevo od svého rodičovského uzlu, pak uložíme nulu a pokud byl uzel vpravo, pak uložíme jedničku. Po dosažení kořene stromu jsme získali i kód hledaného znaku. Tento kód je ale uložen pozpátku, proto jej nyní ještě musíme obrátit.

Dekódování

Dekódování HK probíhá velmi podobně jako jeho samotné kódování a to tak, že v prvním kroku dojde k načtení délky kódovaného streamu a pole četností. V druhém kroku voláním metody *CreateTree* vytvoříme z pole četností Huffmanův strom podle pravidel popsaných výše.

```
//prochazeni vsech znaku
for (int a = 0; a < length & a < input.Length; a++)
{
   byte symbol = input[a];
   byte mask = 128;
   // Procházení jednotlivých bitů
   for (int i = 0; i < 8; i++)
   {
          if (treePeak.isLastNode)
          {
                 // Speciální případ stromu, jen s jedním uzlem
                 listToReturn.Add(treePeak.symbol);
          }
          else  // Běžný stav
          {
                 // Procházení stromu
                 if ((symbol & mask) > 0)
                        DecodeInd = DecodeInd.nodeChilds[1];
                 else
                        DecodeInd = DecodeInd.nodeChilds[0];
                 mask = (byte)(mask >> 1);

                 //nejde o konec stromu?
                 if (DecodeInd.isLastNode)
                 {
                        //Konec stromu
                        listToReturn.Add(DecodeInd.symbol);
                        DecodeInd = treePeak;
                 }
          }
   }
}
```

Zdrojový kód 11: Dekódování pole symbolů pomocí metody *DecodeArray*

Poté v posledním kroku již jen procházíme stromem od kořene k listům tak, že pokud ze zakódovaného streamu načteme znak nula, pak se posouváme na levého potomka, a pokud čteme znak jedna, pak se posouváme na pravého potomka. Ve chvíli, kdy narazíme na list (hodnota proměnné isLastNode bude false), pak jsme našli hledaný znak, ten uložíme a dalším čteným bitem již opět procházíme strom od kořene.

3.1.4 Levensteinovo kódování

Implementace Levensteinova kódování se skládá ze dvou statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream a vrací pole čísel.

Kódování

Kódování obsažené v metodě *EncodeOnlyNumbers* probíhá takto. V prvním kroku, pokud se číslo nerovná nule, se číslo zapíše binárně do proměnné *bin*, zkrátí se o první jedničku a do proměnné *M* se uloží počet zapsaných bitů. Nyní se do doby než proměnná *M* dosáhne nuly, opakuje tento krok s postupným zvyšováním proměnné *C* o jedničku. Následně je před číslo ještě zapsáno *C* jedniček a jedna nula. V posledním kroku je doplněna hlavička obsahující začínající počet prázdných bitových nul.

```
// Načti čísla
int[] nums = UtilityClass.ReadNumbersFromRowFile(stream);

foreach (int num in nums)
{
   int C;
   // Nula je definována jako 0
   if (num == 0)
          C = 0;
   else
          C = 1;
   // Binární číslo bez první jedničky
   string bin = Convert.ToString(num, 2).Substring(1);
   int M = bin.Length;
   sb.Insert(sb_length, bin);

   // Délka binárního zápisu slouží jako další číslo, dokud není
rovno 0

   while (M != 0)
   {
          C++;
          bin = Convert.ToString(M, 2).Substring(1);
          sb.Insert(sb_length, bin);
          M = bin.Length;
   }

   // Na začátek C * 1 oddělených nulou
   sb.Insert(sb_length, '0');
   for (int i = 0; i < C; i++)
          sb.Insert(sb_length, '1');

   while (sb.Length >= 8)
   {
          encodedNumbers.Add(UtilityClass.MakeBinaryStringFrom-
Beginning(sb));
   }

sb_length = sb.Length;
}
```

Zdrojový kód 12: Statická metoda *EncodeOnlyNumbers* Levensteinova kódování

Dekódování

Dekódování v metodě *DecodeOnlyNumbers* probíhá ve třech krocích. V prvním kroku dojde k načtení hlavičky obsahující počet nevyužitých bitů zakódovaného streamu a jejich následné odstranění. V druhém kroku je do proměnné *numBits* uložen počet začátečních jedniček. Nyní hodnotu proměnné *N* nastavíme na jedna. Následně dochází k přečtení vždy *N* bitů a na jejich začátek vložení jedničky a uložení do zpět do proměnné *N*. Tento krok je opakován *numBits-1* krát. Výsledné číslo se nachází v proměnné *N*.

```
List<int> decodedNumbers = new List<int>();

StringBuilder sb = new StringBuilder();
byte[] header = new byte[1];

byte[] encodedNumbers =
UtilityClass.ReadDataFromEncodedFile(stream, 1, ref header);

string decodedString = UtilityClass.ConvertDataToStringOf-
Bits(encodedNumbers, header[0]);

StringReader sr = new StringReader(decodedString);

int bit;
int numberBits = 0;
while ((bit = sr.Read()) != -1)
{
   if (bit == '1')
   {
          numberBits++;
          continue;
   }

   // 0 je definována jako nula
   if (numberBits == 0)
   {
          decodedNumbers.Add(0);
          continue;
   }

   // počtem 1 opakujeme čtení, kde přečtené číslo určuje počet
bitů k přečtení v další iteraci
   int N = 1;
   for (int i = 1; i < numberBits; i++)
   {
          string encoded = "";
          for (int j = 1; j <= N; j++)
                 encoded += (char)sr.Read();
          encoded = "1" + encoded;
          N = Convert.ToInt32(encoded, 2);
   }

   numberBits = 0;
   decodedNumbers.Add(N);
}

return decodedNumbers.ToArray();
```

Zdrojový kód 13: Statická metoda *DecodeOnlyNumbers* Levensteinova kódování

3.1.5 Elias gamma kódování

Implementace Elias gamma kódování se skládá ze dvou statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream a vrací pole čísel.

Kódování

Kódování obsažené v metodě *EncodeOnlyNumbers* probíhá takto. V prvním kroku tvorby kódu dojde k zápisu *n* nul, kde *n* se rovná dvojkovému logaritmu kódovaného čísla. Následně je do kódu připsána binární reprezentace čísla. Postupně je také bitový kód převáděn na bajtový a ten je následně ukládán do pole bajtů. V posledním kroku je doplněna hlavička obsahující začínající počet prázdných bitových nul.

```
StringBuilder sb = new StringBuilder();
List<byte> encodedNumbers = new List<byte>();
byte numberOfEmptyBits;

// Načti čísla
int[] nums = UtilityClass.ReadNumbersFromRowFile(stream);

// Zakódovává steam
foreach (int num in nums)
{
   int l = (int)Math.Log((double)num, 2);
   // počet nul jako logaritmus čísla n
   for (int a = 0; a < l; a++)
          sb.Append(0);

   string binary = UtilityClass.PaddedBin(num, l);

   // První symbol sice má délku jedna, ale je zakodován pouze 1
   if (num != 1)
          sb.Append(binary);

   while (sb.Length >= 8)
          encodedNumbers.Add(UtilityClass.MakeBinaryStringFrom-
Beginning(sb));

}
return encodedNumbers.ToArray();
```

Zdrojový kód 14: Statická metoda *EncodeOnlyNumbers* Elias gamma kódování

Dekódování

Dekódování v statické metodě *DecodeOnlyNumbers* probíhá ve dvou krocích. V prvním kroku dojde k načtení hlavičky obsahující počet nevyužitých bitů zakódovaného streamu a k jejich následnému odstranění. V druhém kroku je při postupném čtení do proměnné *numBits* uložen počet začátečních nul. Následně dochází k postupnému čtení bitů ze streamu a jejich ukládání do proměnné *current*. Počet čtení je roven hodnotě proměnné *numBits-1*. V posledním kroku k proměnné *current* ještě na začátek přidáme jedničku, kterou jsme sice načetli dříve, ale nezapsali. Výsledné číslo se nachází v proměnné *current.*

```
StringReader sr = new StringReader(decodedString);

int numberBits = 0;
int bit;
```

```
while ((bit = sr.Read()) != -1)
{
   if (bit == '0')
   {
          numberBits++;
          continue;
   }

   int current = 0;
   for (int a = numberBits - 1; a >= 0; a--)
   {
          if ((bit = sr.Read()) != -1 && bit == '1')
                current |= 1 << a;
   }
   current |= 1 << numberBits; //První bit byl již přečten dříve

   decodedNumbers.Add(current);
   numberBits = 0;
}

return decodedNumbers.ToArray();
```

Zdrojový kód 15: Statická metoda *DecodeOnlyNumbers* Elias gamma kódování

3.1.6 Elias omega kódování

Implementace Elias omega kódování se skládá ze čtyř statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování pomocí privátní metody *RecursiveEncode*. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream a vrací pole čísel. K dekódování využívá privátní metodu *RecursiveDecode.*

```
private static string RecursiveDecode(string s, int n)
{
   if (s[0] == '0')
   {
          string builder = n.ToString() + " " + s.Substring(1);
          return builder;
   }
   else
   {
          int m = Convert.ToInt32(s.Substring(0, n + 1), 2);
          return RecursiveDecode(s.Substring(n + 1), m);
   }
}
```

Zdrojový kód 16: Metoda *RecursiveDecode* Elias omega kódování

Kódování

Elias omega kódování probíhá rekurzivně voláním metody *RecursiveEncode* a přidáním nuly k výsledku. Jako parametr tato metoda přijímá číslo *num*. Tato metoda buď skončí, pokud hodnota proměnné *num* je menší než jedna, nebo na konec kódovaného řetězce přidá číslo *num* a rekurzivně zavolá metodu *RecursiveEncode* s parametrem binární délky čísla *num-1*. V metodě *EncodeOnlyNumbers* je postupně bitový kód převáděn na bajtový a ten je následně ukládán do pole bajtů. V posledním kroku je doplněna hlavička obsahující začínající počet prázdných bitových nul.

```
private static string RecursiveEncode(int num)
{
   if (num <= 1)
```

```
            return "";
        string binary = Convert.ToString(num, 2);
            return RecursiveEncode(binary.Length - 1) + binary;
}
```

Zdrojový kód 17: Metoda *RecursiveEncode* Elias omega kódování

Dekódování

Dekódování ve statické metodě *DecodeOnlyNumbers* probíhá v těchto krocích. V prvním kroku dojde k načtení hlavičky obsahující počet nevyužitých bitů zakódovaného streamu a jejich následné odstranění. Poté následuje samotné rekurzivní dekódování zavoláním metody *RecursiveDecode* s parametrem *n* se rovná jedna a celým Elias omega kódem. Tato metoda buď skončí a vrátí *n*, pokud je na prvním místě Elias omega kódu nula, nebo přečte *n+1* znaků, uloží je do *n* a rekurzivně zavolá metodu *RecursiveDecode* s parametry zkráceného Elias omega kódu a čísla *n*.

```
string decodedString = UtilityClass.ConvertDataTo-
StringOfBits(encodedNumbers, header[0]);

string rec;
string[] parts = new string[2] { "", "" };
parts[1] += decodedString.ToString();

while (true)
{
   rec = RecursiveDecode(parts[1], 1);
   parts = rec.Split(' ');
   decodedText.Add(Convert.ToInt32(parts[0]));
   if (parts[1] == ""))
         break;
}

return decodedText.ToArray();
```

Zdrojový kód 18: Statická metoda *DecodeOnlyNumbers* Elias omega kódování

3.1.7 Exponenciálně-Golombovo kódování

Implementace Exponenciálně-Golombova kódování se skládá ze dvou statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream a vrací pole čísel.

Kódování

Kódování, které jsem implementoval do metody *EncodeOnlyNumbers* Exponenciálně-Golombova kódování, probíhá ve třech krocích. V prvním kroku se kódované číslo převede do binární podoby a rozdělí se na dvě části podle parametru *k*. K první části se poté přičte jednička a obě části se opět ve stejném pořadí složí dohromady. Nyní se před tento řetězec zapíše tolik nul, jaká byla délka první části. V posledním kroku je opět doplněna hlavička obsahující začínající počet prázdných bitových nul.

```
// Načte čísla
int[] nums = UtilityClass.ReadNumbersFromRowFile(stream);

foreach (int num in nums)
{
   // podle k rozdělíme binární zápis čísla na 2 části
   string binary = Convert.ToString(num, 2);
```

```
   int len = binary.Length;
   string prefix = binary.Substring(0, len - k);
   string suffix = binary.Substring(len - k);

   // k prefixu se přičte 1
   binary = Convert.ToString(Convert.ToInt32(prefix, 2) + 1, 2);

   // na začátek se přidá tolik nul, kolik je délka binary - 1
   for (int i = 1; i < binary.Length; i++)
          sb.Append("0");
   sb.Append(binary);
   sb.Append(suffix);

   while (sb.Length >= 8)
          encodedNumbers.Add(UtilityClass.MakeBinaryStringFrom-
Beginning(sb));
}
return encodedNumbers.ToArray();
```

Zdrojový kód 19: Statická metoda *EncodeOnlyNumbers* Exponenciálně-Golombova kódování

Dekódování

Dekódování Exponenciálně-Golombova kódu probíhá ve statické metodě *DecodeOnlyNumbers* ve čtyřech krocích. V prvním kroku dojde k načtení hlavičky obsahující počet nevyužitých bitů zakódovaného streamu a k jejich následnému odstranění. Ve druhém kroku je při postupném čtení do proměnné *numBits* uložen počet začátečních nul před jedničkou. Následně dochází k postupnému čtení bitů ze streamu a k jejich ukládání do proměnné *current*. Počet čtení je roven hodnotě proměnné *numBits-1*. V dalším kroku k proměnné *current* ještě na začátek přidáme jedničku, kterou jsme sice načetli dříve, ale nezapsali, a poté ještě od proměnné *current* odečteme jedničku. V posledním kroku pak k tomuto číslu zprava přidáme nových *k* přečtených bitů ze streamu.

```
StringReader sr = new StringReader(decodedString);

int numberBits = 0;
int bit;

while ((bit = sr.Read()) != -1)
{
   if (bit == '0')
   {
          numberBits++;
          continue;
   }

   // přečte počet bitů podle úvodních nul
   int current = 0;
   for (int a = numberBits - 1; a >= 0; a--)
   {
          if ((bit = sr.Read()) != -1 && bit == 49)
                 current |= 1 << a;
   }
   current |= 1 << numberBits;
current = current -1;

   // přečte dalších k bitů jako suffix
   string suffix = "";
   for (int a = 0; a < k; a++)
```

```
            suffix += sr.Read();

        // dekrementuje číslo o 1, přiřetězí k bitů a převede na číslo
        decodedNumbers.Add(Convert.ToInt32(Convert.ToString(current,
2) + suffix, 2));
        numberBits = 0;
    }

    return decodedNumbers.ToArray();
```

Zdrojový kód 20: Statická metoda *DecodeOnlyNumbers* Exponenciálně-Golombova kódování

3.1.8 Fibonacciho kódování

Implementace Levensteinvova kódování se skládá ze dvou statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream a vrací pole čísel.

Kódování

Kódování obsažené v metodě *EncodeOnlyNumbers* probíhá takto. V prvním kroku tvorby kódu do proměnné *n* uložíme číslo pro zakódování a do výsledného bitového kódu vložíme jedničku. Poté v kroku druhém nalezneme nejvyšší Fibonacciho číslo, které je menší nebo rovno proměnné *n*. Nyní již procházíme všechna Fibonacciho čísla menší než číslo nalezené v druhém kroku a pokud je toto číslo menší nebo rovno hodnotě *n*, tak jej od čísla *n* odečteme a do výsledného kódu vložíme jedničku, jinak do výsledného kódu vložíme nulu. Po zakódování čísla dochází i k postupnému převádění na bajtový kód, který je ukládán do pole bitů. V posledním kroku je také doplněna hlavička obsahující začínající počet prázdných bitových nul.

```
// Načte čísla
int[] nums = UtilityClass.ReadNumbersFromRowFile(stream);
int fibcounter = 0;
foreach (int num in nums)
{
    int n = num;
    sb.Insert(sb_length, '1');
    // nejvyšší fibonacciho číslo obsažené v n
    while (n >= fib[fibcounter])
            fibcounter++;

// extrahuje fibonacciho čísla, za která píše 1, jinak 0
    for (int i = fibcounter - 1; i >= 2; i--)
    {
            if (n >= fib[i])
            {
                    n -= (int)fib[i];
                    sb.Insert(sb_length, '1');
            }
            else
                    sb.Insert(sb_length, '0');
    }
    fibcounter = 0;

    while (sb.Length >= 8)
            encodedNumbers.Add(UtilityClass.MakeBinaryString-
FromBeginning(sb));
```

```
    sb_length = sb.Length;
}

return encodedNumbers.ToArray();
```

Zdrojový kód 21: Statická metoda *EncodeOnlyNumbers* Fibonacciho kódování

Dekódování

Dekódování Fibonacciho kódu probíhá v statické metodě *DecodeOnlyNumbers*. V prvním kroku dojde k načtení hlavičky obsahující počet nevyužitých bitů zakódovaného streamu a k jejich následnému odstranění. V druhém kroku ukládáme jednotlivé načítané bity kódu do proměnné *decoded* a zároveň hledáme řetězec 11. Po nalezení tohoto řetězce získáme zakódované číslo z proměnné *decoded* sečtením všech Fibonacciho čísel, které jsou v tomto řetězci označeny jedničkou.

```
StringReader sr = new StringReader(decodedString);

int bit;
int number = 0;
string decoded = "";

while ((bit = sr.Read()) != -1)
{
    if (decoded.Length == 0 || decoded[decoded.Length - 1] != '1'
|| bit != 49)
    {
            decoded += (char)bit;
            continue;
    }

    for (int i = 0; i < decoded.Length; i++)
            if (decoded[i] == '1')
                    number += (int)fib[i + 2];

    decodedNumbers.Add(number);
    number = 0;
    decoded = "";
}

return decodedNumbers.ToArray();
```

Zdrojový kód 22: Statická metoda *DecodeOnlyNumbers* Fibonacciho kódování

3.1.9 Simple-9 kódování

Implementace Simple-9 kódování se skládá ze čtyř statických metod. Metoda *EncodeOnlyNumbers* načítá čísla z textového streamu a provádí jejich kódování při využití pomocné metody *useRange*, která zajišťuje správné uložení do vytvořených 32 bitových bloků. Metoda *DecodeOnlyNumbers* načítá zakódovaný stream, dekóduje jej a vrací pole čísel.

```
private static string useRange(int[] nums, int a, int b)
{
    string output = "";

    // počet bitů jednoho čísla
    int len = (int)(28 / (1 + b - a));
    // nepoužité bity
    int unused = 28 % (1 + b - a);
    // pozice podle tabulky
```

```
   int selector = Array.BinarySearch(lengths, len);

   if (selector >= 0)
          // kód podle pozice
          output = UtilityClass.PaddedBin(selector, 4);
   else
          return "";

   for (int i = a; i <= b; i++)
          output += UtilityClass.PaddedBin(nums[i], len);
   for (int i = 0; i < unused; i++)
          output += "0";

return output;
}
```

Zdrojový kód 23: Pomocná metoda *useRange* Simple-9 kódování

Kódování

Kódování obsažené v metodě *EncodeOnlyNumbers* probíhá takto. V prvním kroku se proměnná *max* nastaví na hodnotu prvního čísla. Poté v druhém kroku dochází k porovnávání hodnot postupně načítaných čísel s proměnnou *max* a pokud je hodnota čísla vyšší, zapíše se do proměnné *max*. Zároveň, pokud byla změněna hodnota *max,* nastane přepočet proměnné *dist*, která udává minimální počet bitů potřebných pro zakódování největšího čísla. Ve třetím kroku dochází ke zjišťování, zda již nedošlo k překročení kapacity 28 bitového bloku dat a pokud ano, dojde pomocí metody *useRange* k vytvoření 32 bitového bloku dat, který se skládá z hlavičky obsahující délku datových skupin a binárních čísel délky *dist*. Pokud nedošlo k překročení datové kapacity, pak dochází k opakování kroku dva. Postupně je také bitový kód převáděn na bajtový a ten je následně ukládán do pole bajtů.

```
// Načte čísla
int[] nums = UtilityClass.ReadNumbersFromRowFile(stream);

int max = nums[0];
int j = 0;
int dist = 1;

for (int i = 0; i < nums.Length; i++)
{
   int a = nums[i];
   j++;

   if (a > max) max = a;
          dist = (int)Math.Ceiling(Math.Log(max + 1, 2));

   // pokud se již načítaná čísla nevlezou do 28 bitů, musí se
blok ukončit
   if (j * dist > 28 || i == nums.Length - 1)
   {
          if (j * dist > 28)
          {
                 sb.Append(useRange(nums, i - j + 1, i - 1));
                 j = 1;
          }
          dist = 1;
          max = nums[i];
          if (i == nums.Length - 1)
                 sb.Append(useRange(nums, i - j + 1, i));
```

```
   }

   while (sb.Length >= 8)
          encodedNumbers.Add(UtilityClass.MakeBinaryStringFrom-
Beginning(sb));
}

return encodedNumbers.ToArray();
```

Zdrojový kód 24: Statická metoda *EncodeOnlyNumbers* Simple-9 kódování

Dekódování

Dekódování v statické metodě *DecodeOnlyNumbers* probíhá ve třech krocích. V prvním kroku dojde k načtení 32 bitového bloku dat. Poté dojde k načtení prvních čtyř bitů a správné identifikaci počtu a velikosti *M* zakódovaných skupin. Ve třetím a posledním kroku již dochází k načítání datových bloků délky *M*.

```
string decodedString = UtilityClass.ConvertDataToStringOf-
Bits(encodedNumbers);

int stringLength = decodedString.Length;

int pos = 0;
int inc = 0;
int symbol;
while (pos < stringLength)
{
   // délka načítaných bloků, počet bitů
   inc = lengths[Convert.ToInt32(decodedString.Sub-
string(pos, 4), 2)];
   for (int i = pos + 4; i <= pos + 32 - inc; i += inc)
          if ((symbol = Convert.ToInt32(decodedString.Sub-
string(i, inc), 2)) > 0)
                 decodedText.Add(symbol);
   pos += 32;
}

return decodedText.ToArray();
```

Zdrojový kód 25: Statická metoda *DecodeOnlyNumbers* Simple-9 kódování

4 Testování

Pro testování implementovaných algoritmů mi bylo vedoucím práce poskytnuto 6 testovacích souborů. Pět těchto souborů obsahovalo čísla s uniformním rozložením v rozsahu 2^x až 2^{y-1}. Šestý soubor obsahoval data s Gaussovým rozložením čísel, které je definováno jako $f(x) = (2*pí*s^2)^{-0.5}*exp(-0.5*s^{-2}*(x-m)^2)$, pro m = 0 a s = 128. Ve všech šesti případech soubory obsahovaly jeden milion hodnot.

4.1 Uniformní rozložení

4.1.1 Uniformní rozložení v rozsahu 1 až 31

Čísla v rozsahu $<1;2^5>$ zabírají pouze 5 z 32 bitů, a proto jejich kódování přináší největší přínos ve formě vysokého komprimačního poměru k binárnímu kódu délky 32 bitů. Přesněji, pokud bychom tyto pěti bitová čísla zakódovali do bloků o fixní délce 5, pak bychom dosáhli reálné úspory 84%. Nevýhoda kódování pevné délky však tkví v její nízké flexibilitě a v závislosti délky všech kódu na nejvyšší předem nutně známé hodnotě čísel.

Jak lze vyčíst z tabulky 7, nejvyššího kompresního poměru dosáhlo kódování Simple-9, které překonalo hranici 80 %. Toto vysoké číslo úspory místa bylo dosaženo právě z důvodu, že Simple-9 kódování staví jak na kódováním variabilní, tak i fixní délky. Druhým kódováním s nejvyšším kompresním poměrem bylo Fibonacciho kódování, které dosáhlo úspory 79 % místa. Takto vysoká úspora místa byla dosažena především malým počtem Fibonacciho čísel, ze kterých se čísla v rozmezí 1 až 31 skládají.

Na dalších předních místech se umístily především prefixové kódy, které podobně jako Fibonacciho kódování jsou vysoce efektivní při kódování malých hodnot. Huffmanovo kódování a Redundancy Feedback kódování, které patří do statických, dosáhly úspory 72 % místa. Nejhůře z testovaných algoritmů dopadlo Phased-In kódování, které dosáhlo úspory jen 37 %. Tato nízká efektivita Phased-In kódování i pro rozsah 1 až 31 je dána závislostí délky jednotlivých kódů jen na počtu kódovaných čísel.

4.1.2 Uniformní rozložení v rozsahu 1 až 255

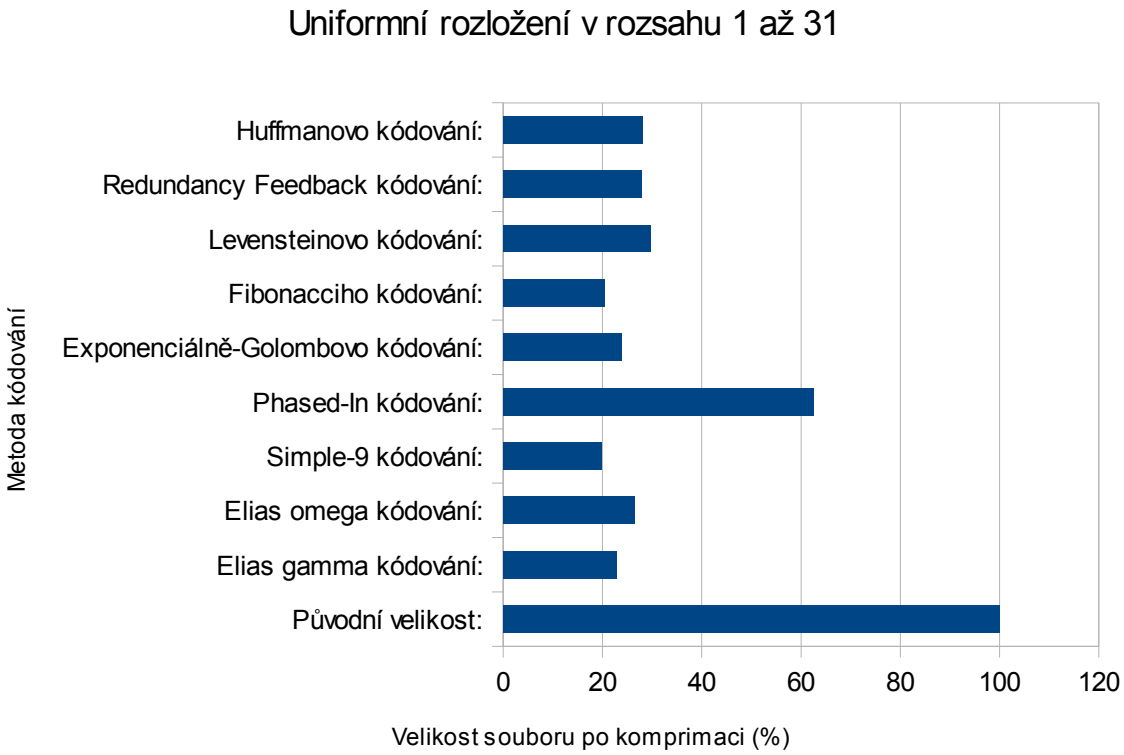
Čísla v rozsahu $<1;2^8>$ zabírají 8 z 32 bitů. Kódováním do bloků fixní délky bychom dosáhli reálné úspory místa 75 %.

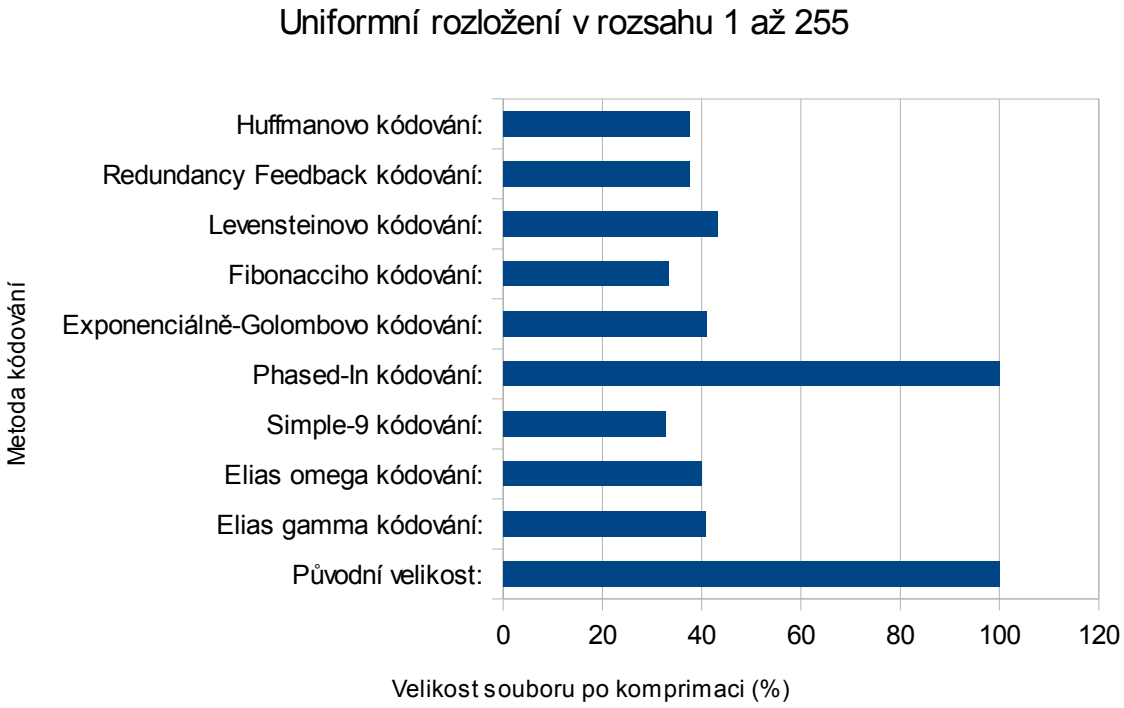
Nejlépe v tomto testu, podobně jako v testu předchozím, obstálo Self-Delimiting kódování Simple-9, které dosáhlo úspory 68% místa. Druhým kódování z nejvyšším kompresním poměrem bylo opět Fibonacciho kódování s úsporou 67% místa.

Na dalších místech se však už umístily ze statistických algoritmů Hufmannovo kódování a Redundancy-Feedback kódování s úsporou 63 %. Hranice úspory 60% pak nedosáhly všechny ostatní prefixové algoritmy a Phased-In kódování dokonce dosáhlo hranice nárůstu délky, protože kódovalo 256 různých řetězců dlouhých 8 bitů svými Phased-In kódy, které tudiž pro hodnotu m = 256 jsou 8 bitů dlouhé.

Tabulka 7:Velikostí jednotlivých zakódovaných souboru

Soubor:	data_00_05.txt		data_00_08.txt		data_08_16.txt		data_16_24.txt		data_24_31.txt		data_norm.txt	
	Počet bajtů	% z 32B	Počet bajtů	% z 32B	Počet bajtů	% z 32B	Počet bajtů	% z 32B	Počet bajtů	% z 32B	Počet bajtů	% z 32B
Původní velikost:	4000000	100	4000000	100	4000000	100	4000000	100	4000000	100	4000000	100
Elias gamma kódování:	915519	22,888	1632865	40,8216	3632751	90,8188	5632821	140,821	7379515	184,488	1524329	38,1082
Elias omega kódování:	1060793	26,5198	1603431	40,0858	2753876	68,8469	4253911	106,348	5127258	128,181	1535434	38,3859
Simple-9 kódování:	797040	19,926	1308184	32,7046	3803284	95,0821	4000000	100	Nelze zakódovat		1250860	31,2715
Phased-In kódování:	2500034	62,5009	4000258	100,006	4000258	100,006	4000258	100,006	4000258	100,006	4000258	100,006
Exponenciálně-Golombovo kódování:	955797	23,8949	1640644	41,0161	3632780	90,8195	5632821	140,821	7379515	184,488	1536114	38,4029
Fibonacciho kódování:	814608	20,3652	1332830	33,3208	2774023	69,3506	4214441	105,361	5473318	136,833	1244460	31,1115
Levensteinovo kódování:	1185793	29,6448	1728431	43,2108	2878876	71,9719	4378911	109,473	5252258	131,306	1660434	41,5109
Redundancy Feedback kódování:	1116998	27,925	1501039	37,526	2495614	62,3904	3412309	85,3077	4012588	100,315	1528548	38,2137
Huffmanovo kódování:	1121960	28,049	1500521	37,513	2496099	62,4025	3428555	85,7139	4000904	100,023	1506201	37,655


Uniformní rozložení v rozsahu 1 až 31
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
Metoda kódování
0
20
40
60
80
100
120
Velikost souboru po komprimaci (%)


Uniformní rozložení v rozsahu 1 až 255
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
Metoda kódování
0
20
40
60
80
100
120
Velikost souboru po komprimaci (%)

4.1.3 Uniformní rozložení v rozsahu 256 až 65535

Čísla v rozsahu $<2^8;2^{16}>$ zabírají 16 z 32 bitů. Kódováním do bloků fixní délky bychom dosáhli reálné úspory místa 50 %.

V tomto testu nejlépe uspěla obě statistická kódování, Redundancy Feedback a Huffmanovo kódování, obě se shodným výsledkem 38% úspory místa oproti původnímu 32 bitovému číslu. Dalším v pořadí jsou dvě prefixová kódování. Fibonacciho kódování, které dosáhlo úspory 31 % a Levensteinovo kódování, které se z předposlední příčky vyhouplo do špičky prefixových kódování. U Fibonacciho kódování došlo k poklesu kompresního poměru z důvodu, že se v něm stále více projevuje častý výskyt binárních čísel obsahujících dvě jedničky na sousedních pozicích.[9]

Kódování Simple-9 si také výrazně pohoršilo z důvodu, že v rozsahu $<2^8;2^{16}>$ dokáže zakódovat jedno až dvě číslo do 28 bitového pole dat. Komprese všech ostatních kódování se pohybují od nuly do deseti procent.

4.1.4 Uniformní rozložení v rozsahu 65536 až 16777215

Čísla v rozsahu $<2^{16};2^{24}>$ zabírají 24 z 32 bitů. Kódováním do bloků fixní délky bychom dosáhli reálné úspory maximálně 25 %.

V tomto rozsahu je již začínají značnou měrou projevovat nedostatky prefixových kódování a průměrná délka kódu všech těchto kódování již přesahuje původní délku 32 bitů. Pro příklad uvádím Exponenciálně-Golombovo a Elias Gamma kódování, která zabírají 140 % původní délky.

Nejvyšší úspory místa opět dosahují Huffmanovo kódování a Redundancy Feedback kódování, které stále komprimují vstupní data o 15 %. Kódování Simple-9 tento rozsah hodnot zakódovává do přesně stejného počtu bitů, protože tvoří 32 bitové bloky s právě jedním blokem obsahujícím číslo. Protože Phased-In kódování není závislé na rozsahu dat, dochází zde k zakódování do stále stejné délky.

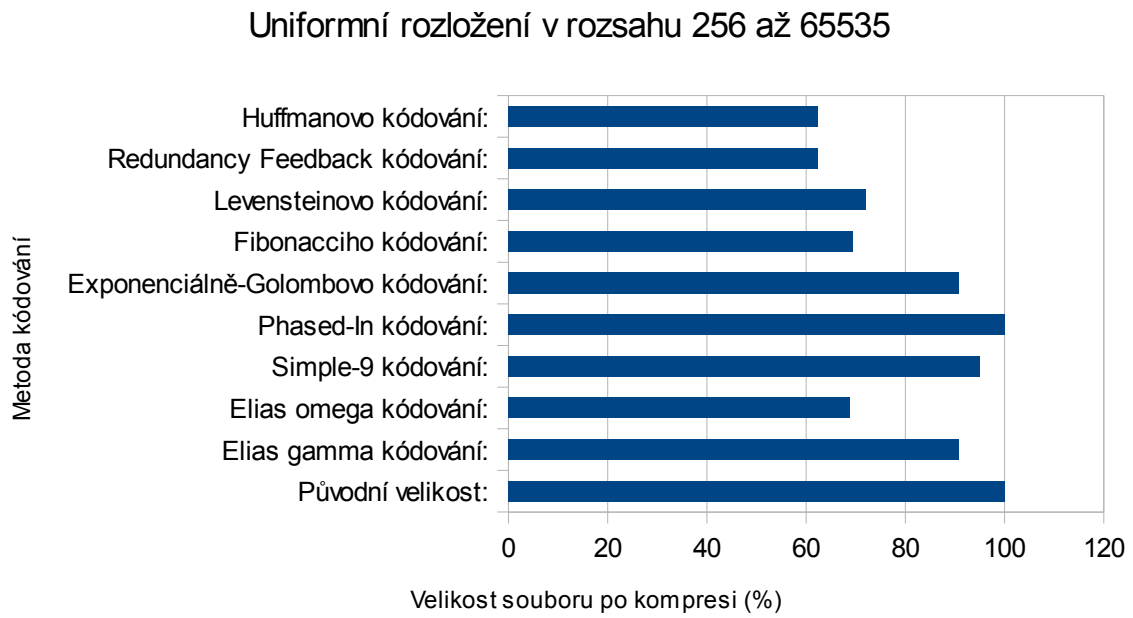
4.1.5 Uniformní rozložení v rozsahu 16777216 až 2147483647

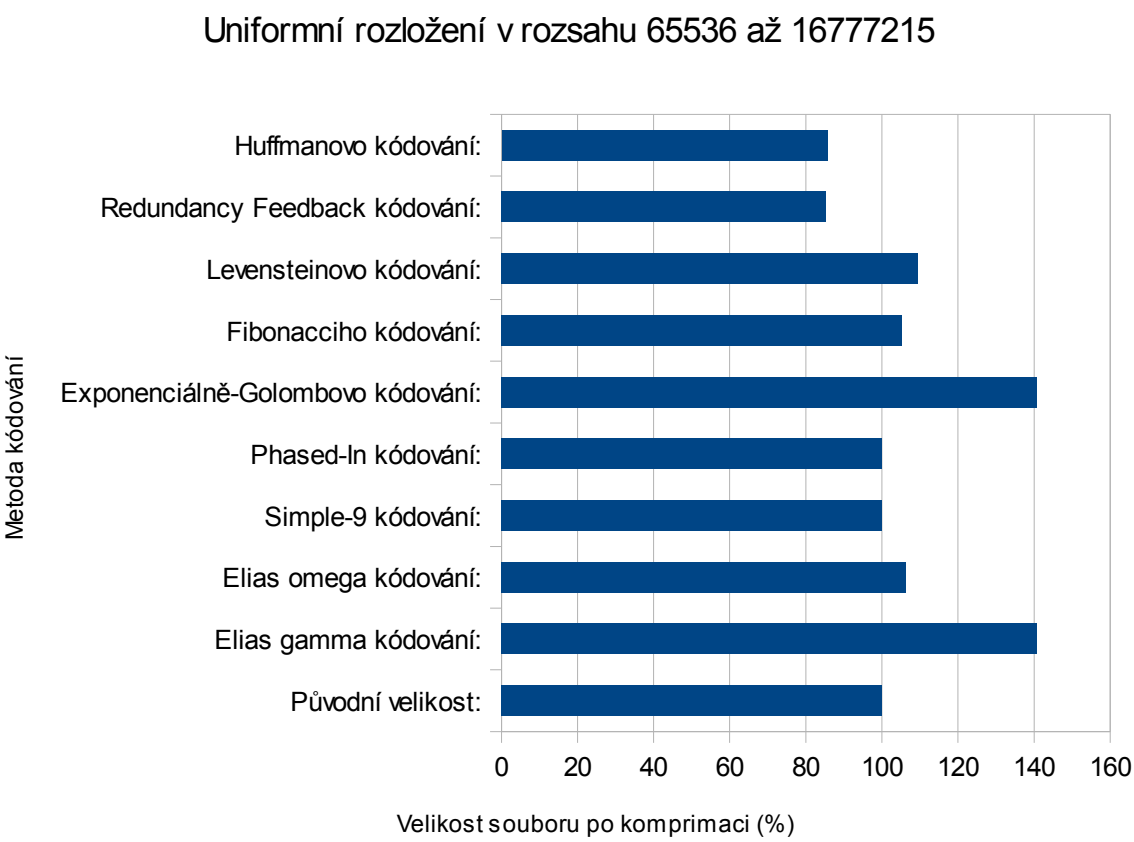
Žádná kódování v rozsahu $<2^{24};2^{32}>$ již nemůžou poskytnout úsporu místa. Nejefektivnější se v tomto rozsahu jeví Phased-In kódování, které spolu s Huffmanovým kódováním a Redundancy Feedback kódováním přesáhly stoprocentní hranici jen o několik tisícin až desetin procenta.

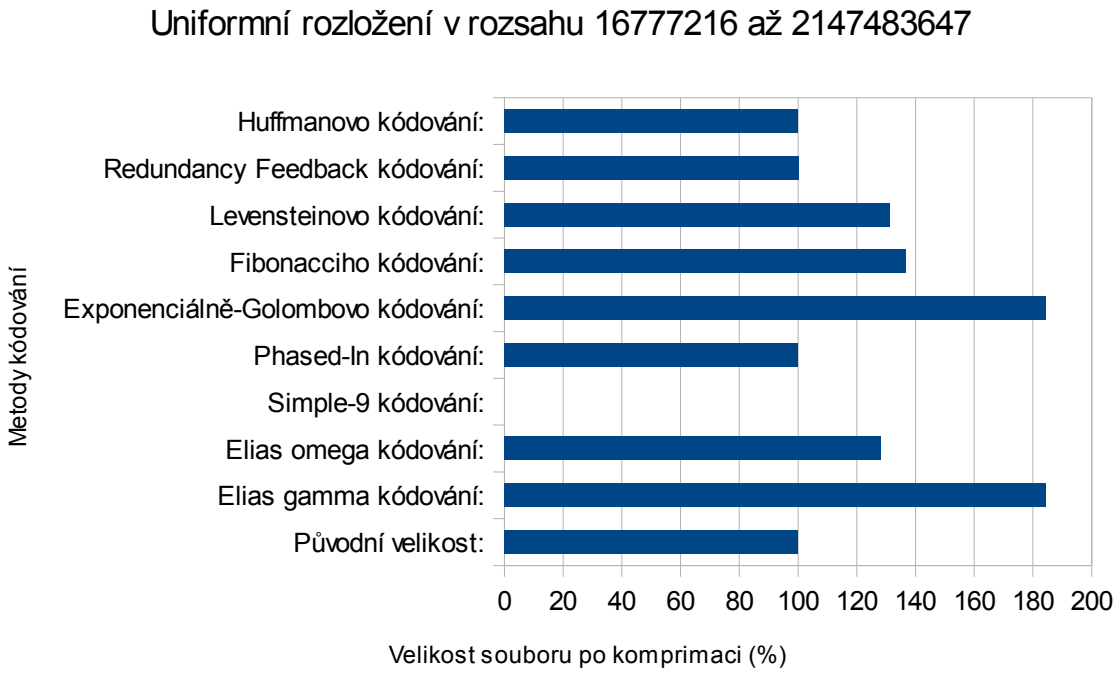
Kódování Simple-9 v tomto rozsahu již nemůžeme aplikovat, protože jeho blok dat má maximální délku 28 bitů. Pro tento rozsah jsou prefixová čísla již vysoce neefektivní, protože jejich kódováním dochází k nárustu kódu až o 85 %.

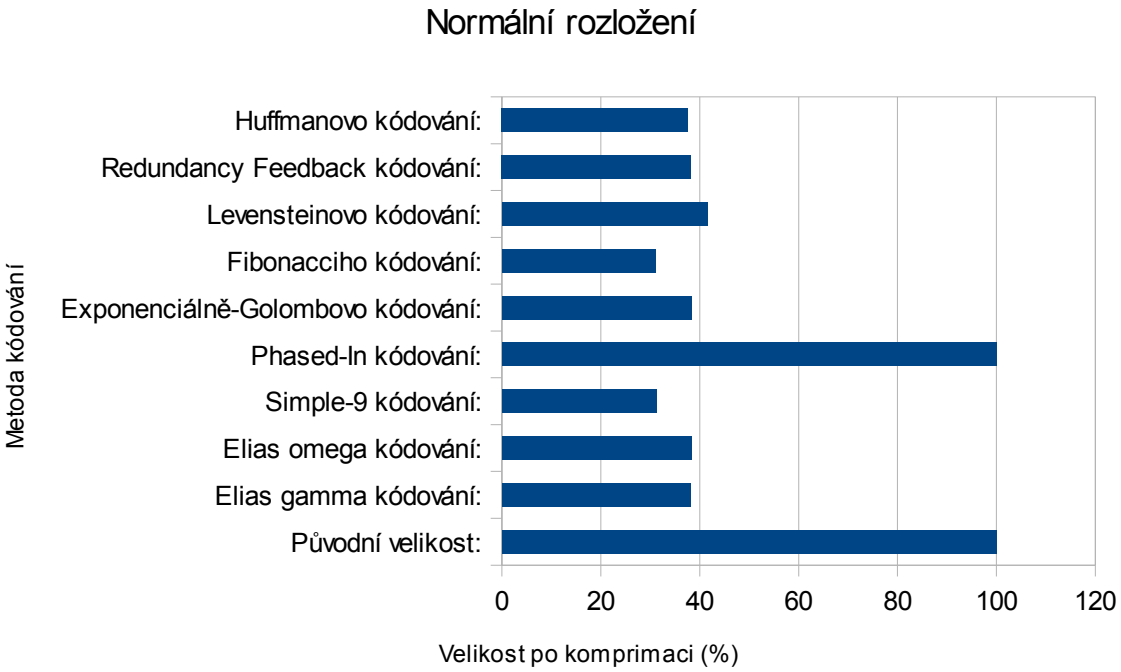
4.2 Normální rozložení

Při testování komprese dat nad normálním, Gaussovým rozložením čísel prokázala všechna kódování až na Phased-In kódování dobrou schopnost komprese. U těchto algoritmů se úspora místa pohybovala od 70 % do 60 %. Nejlépe v tomto rozsahu obstály Fibonacciho a Simple-9 kódování


Uniformní rozložení v rozsahu 256 až 65535
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
Metoda kódování
0
20
40
60
80
100
120
Velikost souboru po kompresi (%)


Uniformní rozložení v rozsahu 65536 až 16777215
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
Metoda kódování
0
20
40
60
80
100
120
140
160
Velikost souboru po komprimaci (%)


Uniformní rozložení v rozsahu 16777216 až 2147483647
Metody kódování
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
0 20 40 60 80 100 120 140 160 180 200
Velikost souboru po komprimaci (%)


Normální rozložení
Metoda kódování
Huffmanovo kódování:
Redundancy Feedback kódování:
Levensteinovo kódování:
Fibonacciho kódování:
Exponenciálně-Golombovo kódování:
Phased-In kódování:
Simple-9 kódování:
Elias omega kódování:
Elias gamma kódování:
Původní velikost:
0 20 40 60 80 100 120
Velikost po komprimaci (%)

5 Závěr

V rámci této práce jsem se detailně seznámil s různými metodami kódování znaků a čísel v rozsahu $< 1; 2^{32} >$ do bitové podoby. Tyto své poznatky jsem uplatnil v rámci implementace kompresního frameworku. Při implementaci jsem se snažil využívat pokročilých a efektivních přístupů, ale zároveň jsem se snažil udržet zdrojový kód krátký a tudíž i dobře čitelný. Pokud to bylo přínosné, tak jsem jej opatřil řádkovými komentáři.

Phased-In kódování vykazovalo po celou dobu testovaní špatné výsledky komprese, protože na rozdíl od ostatních kódů, není závislé na rozsahu kódovaných čísel, ale na počtu těchto od sebe odlišných kódovaných čísel. Phased-In kódování se tedy vyplatí implementovat pouze pokud jsme přesvědčení, že variabilita dat ke kompresi bude nízká.

Redundancy Feedback kódování je sice běhově jedno z nejnáročnějších na hardwarové prostředky, ale odpovídá mu i kompresní poměr a úspora místa. Zároveň lze na tomto kódu pozorovat vysokou formu flexibility tvorby kódu a tudíž i vysoký rozsah hodnot, při kterých je kódování stále efektivní a dochází ke kompresi dat. Problémem tohoto kódování může být právě jeho postup kódování, které probíhá v obráceném směru, protože ne vždy lze tímto směrem bajty číst.

Simple-9 kódování patřící mezi Self-Delimiting kódování dosáhlo pro čísla do 2^8 dvakrát dvakrát nejlepších výsledků. Avšak pro čísla delší než 9 či 14 bitů se tento algoritmus stal velmi málo účinným a čísla větší než 2^{28} již nedokázal ani kódovat.

Všechna **prefixová kódování** s výjimkou Fibonacciho kódování si v testech vedla obdobně a dosahovala dobrých výsledků pro čísla z intervalů do 2^{16}. I když výhodou těchto kódování je možnost je nasadit i v situacích, kdy nic nevíme o nejvyšších hodnotách, nejvyšší komprese dosahují pro malá přirozená čísla.

U Fibonacciho kódování jsem pozoroval vysokou efektivitu především pro malá čísla, kdy u testovacího souboru data_00_05.bin dosáhlo téměř osmdesáti procentní komprese v porovnání s fixním 32 bitovým kódováním. Se zvyšujícím se rozsahem vstupních čísel však efektivita Fibonacciho kódování klesá a pro vstupní čísla v rozsahu 2^{16} až 2^{24} vykazuje již nárůst délky v porovnání s 32 bitovým kódováním.

Huffmanovo kódování dosahuje v testech obdobné výsledky jako Redundancy Feedback kódování, ale dokáže toho dosáhnout i za cenu menší hardwarové náročnosti. Toto kódování totiž jen na začátku postaví strom z četností výskytů a pak jej při kódování a dekódování pouze prochází. Výhodou Huffmanova kódování je také kódování dat v přirozeném směru.

V průběhu testování se ukázalo, že statistické metody sice pro čísla malá nedosahují tak dobrých výsledků jako některá prefixová kódování, ale, za cenu vyšší hardwarové náročnosti a nutnosti před kódováním znát všechny kódované symboly, pro čísla v jakémkoliv rozsahu poskytují alespoň výsledky srovnatelné s fixním 32 bitovým kódováním.

6 Seznam použité literatury

[1] SALOMON, David. *Variable-length Codes for Data Compression*. 1. vyd. London: Springer-Verlag London Limited, 2007. 196 s. ISBN 978-1-84628-958-3

[2] Přispěvatelé Wikipedie, *Huffman coding* [online], Wikipedie: Otevřená encyklopedie, c2011, Datum poslední revize 7. 3. 2011, 20:39 UTC, [citováno 5. 05. 2011] <http://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=417660352>

[3] SALOMON, David. *Data Compression: The Complete Reference*. 3. vyd. New York: Springer-Verlag New York, Inc, 2004. 899 s. ISBN 0-387-40697-2. Kapitoly 2.6, 2.7 a 2.8, s 66–89.

[4] MACKAY, J.C. David. *Information Theory, Inference, and Learning Algorithms*. verze 7.2. Cambridge: Cambridge University Press. 2005. 628 s. Dostupná z <http://www.inference.phy.cam.ac.uk/mackay/itila/>. Kapitola 5. s 90–108.

[5] Přispěvatelé Wikipedie, *Huffmanovo kódování* [online], Wikipedie: Otevřená encyklopedie, c2010, Datum poslední revize 5. 12. 2010, 13:33 UTC, [citováno 8. 05. 2011] <http://cs.wikipedia.org/w/index.php?title=Huffmanovo_k%C3%B3dov%C3%A1n%C3%AD&oldid=6162947>

[6] GILBERT, E. N. – MOORE E. F. Variable Length Binary Encodings. In *Bell Systém Technical Journal*. Číslo 38. Alcatel-Lucent, 1959. s. 933–967.

[7] BALDWIN, Rich. *Information Theory and Creationism* [online], Algorithmic Information Theory (Chaitin, Solomonoff & Kolmogorov). c2005, Datum poslední revize 14. 7. 2005, [citováno 30. 6. 2011] <http://www.talkorigins.org/faqs/information/algorithmic.html>

[8] SOLOMON, David. *Handbook of Data Compression.* 5. vyd. London: Springer-Verlag London Limited, 2010. 1360 s. ISBN 978-1-84882-902-2. Kapitoly 2, 2.1, 2.2, 2.3, 2.9, 2.10, 2.12, 3.4, 3.6, 3.20

[9] MICHÁLEK, Radek. *Implementace kódů proměnné délky pro celá čísla*. Ostrava, 2010. 38 s. Bakalářská práce na Vysoké škole Báňské – Technické univerzitě Ostrava. Vedoucí diplomové práce Ing. Janu Platoši, Ph.D.